Optimal Global Instruction Scheduling for the Itanium$^{®}$ Processor Architecture

Dissertation

zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften (Dr.-Ing.) der
Naturwissenschaftlich-Technischen Fakultäten der
Universität des Saarlandes

von
Diplom-Informatiker

Sebastian Winkel

Saarbrücken
September 2004

Das Bildmaterial für den Umschlag und für die Abbildungen auf den Seiten 32, 35 und 42 wurde freundlicherweise zur Verfügung gestellt von Intel Deutschland GmbH.

Tag des Kolloquiums:	16.12.2004
Dekan:	Prof. Dr. Jörg Eschmeier
Prüfungsausschuss:	
Gutachter:	Prof. Dr. Reinhard Wilhelm
	Priv.-Doz. Dr. Friedrich Eisenbrand, Max-Planck-Institut für Informatik, Saarbrücken
	Prof. Dr. Peter Marwedel, Universität Dortmund
Vorsitzender:	Prof. Dr. Raimund Seidel
Akademischer Mitarbeiter:	Dr.-Ing. Stephan Thesing

Abstract

On the Itanium 2 processor, effective global instruction scheduling is crucial to high performance. At the same time, it poses a challenge to the compiler: This code generation subtask involves strongly interdependent decisions and complex trade-offs that are difficult to cope with for heuristics. We tackle this $\mathcal{NP}$-complete problem with *integer linear programming (ILP)*, a search-based method that yields provably optimal results. This promises faster code as well as insights into the potential of the architecture. Our ILP model comprises global code motion with compensation copies, predication, and Itanium-specific features like control/data speculation.

In integer linear programming, *well-structured* models are the key to acceptable solution times. The feasible solutions of an ILP are represented by integer points inside a polytope. If all vertices of this polytope are *integral*, then the ILP can be solved in polynomial time. We define two subproblems of global scheduling in which some constraint classes are omitted and show that the corresponding two subpolytopes of our ILP model are integral and polynomial sized. This substantiates that the found model is of high efficiency, which is also confirmed by the reasonable solution times.

The ILP formulation is extended by further transformations like *cyclic code motion*, which moves instructions upwards out of a loop, circularly in the opposite direction of the loop backedges. Since the architecture requires instructions to be encoded in fixed-sized bundles of three, a *bundler* is developed that computes bundle sequences of minimal size by means of precomputed results and dynamic programming.

Experiments have been conducted with a *postpass tool* that implements the ILP scheduler. It parses assembly procedures generated by Intel's Itanium compiler and reschedules them as a whole. Using this tool, we optimize a selection of hot functions from the SPECint 2000 benchmark. The results show a significant speedup over the original code.

Zusammenfassung

Globale Instruktionsanordnung hat beim Itanium-2-Prozessor großen Einfluß auf die Leistung und stellt dabei gleichzeitig eine Herausforderung für den Compiler dar: Sie ist mit zahlreichen komplexen, wechselseitig voneinander abhängigen Entscheidungen verbunden, die für Heuristiken nur schwer zu beherrschen sind. Wir lösen dieses $\mathcal{NP}$-vollständige Problem mit ganzzahliger linearer Programmierung (ILP), einer suchbasierten Methode mit beweisbar optimalen Ergebnissen. Das ermöglicht neben schnellerem Code auch Einblicke in das Potential der Itanium-Prozessorarchitektur. Unser ILP-Modell umfaßt globale Codeverschiebungen mit Kompensationscode, Prädikation und Itanium-spezifische Techniken wie Kontroll- und Datenspekulation.

Bei ganzzahliger linearer Programmierung sind *wohlstrukturierte* Modelle der Schlüssel zu akzeptablen Lösungszeiten. Die zulässigen Lösungen eines ILPs werden durch ganzzahlige Punkte innerhalb eines Polytops repräsentiert. Sind die Eckpunkte dieses Polytops ganzzahlig, kann das ILP in Polynomialzeit gelöst werden. Wir definieren zwei Teilprobleme globaler Instruktionsanordnung durch Auslassung bestimmter Klassen von Nebenbedingungen und beweisen, daß die korrespondierenden Teilpolytope unseres ILP-Modells ganzzahlig und von polynomieller Größe sind. Dies untermauert die hohe Effizienz des gefundenen Modells, die auch durch moderate Lösungszeiten bestätigt wird.

Das ILP-Modell wird um weitere Transformationen wie *zyklische Codeverschiebung* erweitert; letztere bezeichnet das Verschieben von Befehlen aufwärts aus einer Schleife heraus, in Gegenrichtung ihrer Rückwärtskanten. Da die Architektur eine Kodierung der Befehle in Dreierbündeln fester Größe vorschreibt, wird ein *Bundler* entwickelt, der Bündelsequenzen minimaler Länge mit Hilfe vorberechneter Teilergebnisse und dynamischer Programmierung erzeugt.

Für die Experimente wurde ein Postpassoptimierer erstellt. Er liest von Intels Itanium-Compiler erzeugte Assemblerroutinen ein und ordnet die enthaltenen Instruktionen mit Hilfe der ILP-Methode neu an. Angewandt auf eine Auswahl von Funktionen aus dem Benchmark SPECint 2000 erreicht der Optimierer eine signifikante Beschleunigung gegenüber dem Originalcode.

Extendend Abstract

HP and Intel's Itanium Processor Family (IPF) is considered as one of the most challenging processor architectures to generate code for. Its performance is highly compiler-dependent since it relies on static instruction scheduling. The code generator is responsible for extracting a high amount of instruction-level parallelism and exposing it to the processor. To achieve this, it must combine global instruction scheduling (with code motion between basic blocks) with the use of predication and Itanium-specific features like explicit speculation.

When applying these techniques, the compiler must deal with strongly interdependent decisions and keep the increasing resource pressure as a result of compensation copies and speculative computations under control: Overuse can spoil the benefit due to execution unit shortage, but the opposite, a too conservative application, can lead to many unused execution slots. It is difficult for the greedy scheduling heuristics found in Itanium compilers to find a balance. Moreover, resource-constrained instruction scheduling is—even when limited to basic blocks—an $\mathcal{NP}$-complete problem, for which heuristics deliver only approximations. It is unclear how far away these are from exact, optimal solutions.

In this thesis we tackle the *global instruction scheduling problem* for the Itanium 2 processor with *integer linear programming (ILP),* a search-based combinatorial optimization method that yields provably optimal results. Our ILP model comprises global code motion with automated generation of compensation code, predication, as well as vital IPF features like control and data speculative loads. The ILP approach can—within certain limits—resolve the interdependences between the involved decisions and deliver the global optimum in the form of a schedule with minimal length. This promises faster code, as well as some theoretically well-founded insights into the potential of the architecture.

In integer linear programming, the search space of feasible solutions is represented by the integer points inside a polytope. A *well-structured* model, in which these integer points are tightly enclosed by the polytope, is the key to acceptable solution times. If all vertices of the polytope are *integral*, then the ILP can even be solved in polynomial time. Since we are modeling an $\mathcal{NP}$-complete problem, however, we cannot expect to find such a polytope of polynomial size for the entire global instruction scheduling problem, but only possibly for *subproblems*. Therefore we define several such subproblems in which some constraint classes are omitted (such as the resource constraints, which model the occupation of execution units, or the precedence constraints, which enforce data dependence preservation). For subproblems that are no longer $\mathcal{NP}$-complete, ILP formulations may exist that describe integral and polynomial sized subpolytopes—and are in fact developed in this thesis.

For one of these subproblems, this is achieved by reducing it to a *node packing problem* on a perfect graph and exploiting a standard result in order to obtain an integral subpolytope. A further subproblem is modeled as a *network flow problem*, a problem class for which integral polytopes are well known, too. In both cases, it is necessary to reduce the complexity of the resulting ILP formulation afterwards, which is in the former case even exponential. This is done by removing redundant constraints and by applying *integrality-preserving* transformations. These include *lifting* the polytope to a higher dimension (to reduce the number of constraints needed to describe it), or inversely, *projecting* it onto a lower-dimensional subspace (to reduce the number of needed variables).

As a central theoretical result of this thesis, we further show that the identified integral and polynomial sized subpolytopes are *maximal* in the sense that they cannot be extended by other constraint classes without losing one of their two efficiency properties (integrality or polynomial size). This follows—under the assumption $\mathcal{P} \neq \mathcal{NP}$—from two $\mathcal{NP}$-completeness proofs for the extended subproblems. It substantiates that the found ILP model is close to maximal efficiency. The polyhedral analysis also reveals in detail that there is a wide complexity gap between local and global instruction scheduling.

The developed basic ILP model is enriched with further variants of code motion and speculation: *Predicated code motion* extends the scope of code motion via predication. Similarly, *partial-ready code motion* is included, which allows instructions to be moved further upwards along a control flow path by speculatively ignoring data dependences on instructions from other paths. To increase the scheduling scope in the presence of loops, the model is extended to support *code motion into and out of loops*. Upward code motion out of loops includes *cyclic code motion* circularly in the opposite direction of the backedges, a variant which can effectively reduce the schedule length of the loop body. A further add-on models the changes in the branch structure resulting from blocks that are emptied via code motion. On the Itanium architecture, instructions have to be encoded in fixed-sized bundles of three, which are further specified by *templates*. To bundle the obtained schedules, a method is developed that computes bundle sequences of minimal size by means of precomputed results and dynamic programming.

The *experiments* were conducted with a *postpass tool* that implements the ILP scheduler. It parses assembly procedures and optimizes them as a whole. Various precomputations are performed to reduce the search space. During the ILP generation, constraints are tightened dynamically. This leads, in combination with the high efficiency of the used ILP model, to reasonable solution times of not more than a few minutes. Using this tool, we optimize a selection of hot functions from the SPECint 2000 benchmark. Assembly versions of them are generated with Intel's Itanium compiler and fed into the postpass optimizer. The rescheduled code turns out to run significantly faster than the original version.

Although the ILP method is, because of the comparatively long solution times, not suited for product compilers, the experimental results show that it is promising as a stand-alone *optimization tool* for compute-intensive application kernels like compression and encryption routines. A further interesting application is as a *research tool*: The comparison of the optimal results with those of heuristics can help find and quantify room for improvements in the latter. Moreover, the ILP approach can be used to explore the potential of the Itanium architecture since it delivers—in contrast to all heuristics and limit studies—concretely the best achievable solutions.

Ausführliche Zusammenfassung

Bei der Itanium-Prozessorfamilie von HP und Intel gilt Codeerzeugung als eine gleichermaßen wichtige und schwierige Compilerphase. Da diese Prozessorarchitektur auf statischer Instruktionsanordnung beruht, ist ihre Leistung stark von der Qualität des Codeerzeugers abhängig. Dieser ist dafür verantwortlich, ein hohes Maß an Parallelität auf Befehlsebene zu extrahieren und explizit an den Prozessor zu übermitteln. Zu diesem Zweck muß er globale Instruktionsanordnung (mit Codeverschiebungen zwischen Basisblöcken) mit dem Einsatz von Prädikation und Spekulation kombinieren.

Dabei muß er zahlreiche wechselseitig voneinander abhängige Entscheidungen überblicken und insbesondere den erhöhten Ressourcenbedarf infolge von Kompensationscode und spekulativen Berechnungen unter Kontrolle halten: Ein übermäßiger Gebrauch von Codeverschiebungen und Spekulation durch den Compiler kann zu Knappheit von funktionalen Einheiten führen, die den Nutzen wieder zunichte macht. Das Gegenteil, eine zu konservative Benutzung, läßt das Potential der Architektur ungenutzt. Für die in Itanium-Compilern eingesetzten Heuristiken ist es schwer, hier einen Ausgleich zu finden. Darüber hinaus ist Instruktionsanordnung unter begrenzten funktionalen Ressourcen – auch wenn lokal auf einzelne Basisblöcke beschränkt – ein $\mathcal{NP}$-vollständiges Problem, für das Heuristiken nur Näherungslösungen berechnen. Wie weit diese beim Itanium vom globalen Optimum entfernt liegen, ist unbekannt.

In der vorliegenden Arbeit wird das Problem globaler Instruktionsanordnung für den Itanium-2-Prozessor mit *ganzzahliger linearer Programmierung (ILP)* gelöst, einer suchbasierten Methode zur exakten Lösung kombinatorischer Optimierungsprobleme. Das entwickelte ILP-Modell umfaßt globale Codeverschiebungen mit automatischer Einfügung von Kompensationskopien, Prädikation sowie den Einsatz Itanium-spezifischer kontroll- und datenspekulativer Ladebefehle. Der ILP-Ansatz kann – innerhalb gewisser Grenzen – die wechselseitigen Abhängigkeiten zwischen den modellierten Entscheidungen auflösen und das globale Optimum in Form eines Schedules mit minimaler Länge berechnen. Das läßt neben schnellerem Code auch theoretisch fundierte Einblicke in das Potential der Itanium-Prozessorarchitektur erwarten.

Der Suchraum eines ILPs wird durch ganzzahlige Punkte innerhalb eines Polytops repräsentiert. *Wohlstrukturierte* Modelle, bei denen diese Punkte so eng wie möglich von dem Polytop umschlossen werden, sind der Schlüssel zu akzeptablen Lösungszeiten. Ganzzahlige Polytope (mit ausschließlich ganzzahligen Eckpunkten) erlauben sogar Lösbarkeit in Polynomialzeit. Da wir ein $\mathcal{NP}$-vollständiges Problem modellieren, können wir zwar nicht erwarten, ein solches Polytop polynomieller Größe für das Gesamtproblem zu finden, möglicherweise aber für Teilprobleme. Daher definieren wir mehrere solcher Teilprobleme durch Auslassung bestimmter Klas-

sen von Nebenbedingungen (wie zum Beispiel der Ressourcenschranken, die die Belegung der funktionalen Einheiten modellieren, oder der Vorrangbedingungen, die die Einhaltung von Datenabhängigkeiten sicherstellen). Für diejenigen Teilprobleme, die nicht mehr $\mathcal{NP}$-vollständig sind, können ILP-Formulierungen polynomieller Größe existieren, die ganzzahlige Polytope beschreiben – und werden in der Tat in dieser Arbeit entwickelt.

Für eines dieser Teilprobleme geschieht dies durch Reduktion auf ein *Node-Packing-Problem* auf einem perfekten Graphen. Unter Ausnutzung eines bekannten Satzes kann dann eine Beschreibung dieses Problems durch ein ganzzahliges Polytop gefunden werden. Ein weiteres Teilproblem wird als *Netzwerk-Fluß-Problem* formuliert, eine Problemklasse, für die ebenfalls ganzzahlige Polytope wohlbekannt sind. In beiden Fällen ist es allerdings notwendig, die Komplexität der erhaltenen ILP-Formulierungen anschließend zu reduzieren (im ersteren Fall ist sie anfangs sogar exponentiell). Dies geschieht durch Entfernung redundanter Nebenbedingungen und durch die Anwendung von Transformationen auf die Polytope, wie zum Beispiel ein *Lifting* in eine höhere Dimension (um die Anzahl der zur Beschreibung notwendigen Ungleichungen zu reduzieren), oder umgekehrt, eine *Projektion* auf einen Unterraum geringerer Dimension (um die Anzahl benötigter Variablen zu reduzieren). Diese Transformationen werden so durchgeführt, daß sie die Ganzzahligkeit der Polytope bewahren.

Als ein zentrales theoretisches Ergebnis dieser Arbeit zeigen wir dann, daß die gefundenen ganzzahligen Teilpolytope polynomieller Größe *maximal* sind in dem Sinne, daß eine Hinzunahme anderer Klassen von Nebenbedingungen nicht ohne einen Verlust von Ganzzahligkeit oder polynomieller Größe möglich wäre. Dies folgt – unter der Annahme $\mathcal{P} \neq \mathcal{NP}$ – aus zwei $\mathcal{NP}$-Vollständigkeitsbeweisen für die auf diese Weise erweiterten Teilprobleme. Es untermauert die hohe Effizienz der gefundenen Formulierung. Die Ergebnisse der Analyse zeigen außerdem im Detail auf, daß globale Instruktionsanordnung auch aus komplexitätstheoretischer Sicht ein wesentlich schwierigeres Problem als die lokale Variante ist.

Das entwickelte Basismodell wird um weitere Varianten von Codeverschiebungen und Spekulation erweitert: Prädikative Codeverschiebung erweitert den Spielraum für die Plazierung von Befehlen mit Hilfe von Prädikation. Demselben Zweck dient *partial-ready code motion*, das es ermöglicht, Befehle auf einem Kontrollflußpfad dadurch weiter nach oben zu verschieben, indem Datenabhängigkeiten von Befehlen auf anderen Pfaden spekulativ ignoriert werden. Um den Spielraum für die Anordnung von Befehlen bei Anwesenheit von Schleifen zu vergrößern, wird das Modell um Codeverschiebungen *in und aus Schleifen* erweitert. Diese umfassen auch *zyklische Codeverschiebung*, das Verschieben von Befehlen aufwärts über den Schleifenkopf hinaus in Gegenrichtung der Rückwärtskanten der Schleife. Weitere ILP-Formulierungen modellieren Änderungen der Struktur der Sprungbefehle, die sich ergeben, wenn Blöcke durch globale Codeverschiebungen ganz geleert werden.

Die Itanium-Architektur schreibt eine Kodierung der Befehle in *Dreierbündeln* fester Größe vor, deren Befehlstypen durch Auswahl einer Vorlage (*template*) jeweils genauer spezifiziert werden. Um die berechneten Schedules entsprechend zu bündeln, wird eine Methode entwickelt, die Bündelsequenzen minimaler Länge mit Hilfe vorberechneter Teilergebnisse und dynamischer Programmierung erzeugt.

Für die Experimente wurde ein *Postpassoptimierer* erstellt. Er liest von Intels Itanium-Compiler erzeugte Assemblerroutinen ein und optimiert sie als Ganzes, ordnet die enthaltenen Instruktionen also mit Hilfe der ILP-Methode neu an. Die hohe Effizienz der verwendeten Formulierung führt in Kombination mit Vorberechnungen zur Reduktion des Suchraumes sowie einer dynamischen Straffung von Ungleichungen während der ILP-Generierung zu moderaten Lösungszeiten von wenigen Sekunden bis Minuten. Auf einer Auswahl von Funktionen aus dem Benchmark SPECint 2000 erreicht der Optimierer eine signifikante Beschleunigung gegenüber dem Originalcode.

Zwar ist der ILP-Ansatz wegen der vergleichsweise langen Lösungszeiten nicht für klassische Compiler geeignet, die experimentellen Ergebnisse lassen ihn aber hochinteressant als Optimierungswerkzeug für kleine, rechenintensive Softwarekomponenten wie Kompressions- oder Verschlüsselungsroutinen erscheinen. Weitere Einsatzmöglichkeiten ergeben sich in Forschung und Entwicklung in den Bereichen Compiler- und Prozessordesign: Durch den Vergleich mit optimalen Ergebnissen kann Spielraum für Verbesserungen bei den Heuristiken eines Codeerzeugers lokalisiert und quantifiziert werden. Außerdem kann die Methode dazu genutzt werden, das Potential der Itanium-Architektur präzise auszuloten, denn im Gegensatz zu allen Heuristiken und Abschätzungen liefert sie konkret bestmögliche Lösungen.

Acknowledgments

First of all, I would like to thank my advisor Prof. Reinhard Wilhelm for his continuous guidance and support throughout my graduate study. He always found the time to give me his counsel when I needed it. Several invitations to Dagstuhl Seminars allowed me to learn to know other researchers and to present my work to them. He created an open, optimistic working atmosphere in his group that inspired my research.

I enjoyed working with my pleasant and humorous colleagues. My thanks go especially to Stephan Thesing for his unflagging helpfulness, and to Ingmar Stein, who did an excellent job in implementing the bundler.

I am grateful to the Deutsche Forschungsgemeinschaft for supporting this research by a graduate fellowship under the graduate studies program "Leistungsgarantien für Rechnersysteme" at Saarland University. In particular, I wish to thank the former speaker of the program, Prof. Dr.-Ing. Gerhard Weikum, and the current speaker, Prof. Dr. Raimund Seidel.

My special thanks go to Mary Lou Soffa and Keith D. Cooper for their encouragement and support. Also, I like to thank Prof. Dr. Kurt Mehlhorn and the Max-Planck-Institut für Informatik, Saarbrücken, for granting access to their CPLEX installation over a long period of time.

Finally, I would like to thank my parents Marie and Albert and my brother Georg for their patience and their steadfast support throughout the years.

Contents

Chapter 1

Introduction

In 1994, Intel and Hewlett-Packard began codeveloping a radically new 64-bit computer architecture that became known as the Itanium Processor Family (IPF) [Alp03]. Ten years and an estimated $5 billion later [ML02], the second-generation Itanium 2 processor has entered the markets for high-end servers and workstations. What makes the design approach groundbreaking—and to some observers also arguable—is that it marks a radical departure from prevailing superscalar CISC and RISC architectures, especially with regard to the division of responsibilities between processor and compiler. In addition, it introduces new concepts from academic and corporate research like explicit speculation that aim at increasing the instruction throughput beyond RISC [SRM$^+$94]. The following introduction gives an understanding of the rationales behind the architecture; it uses several terms that are explained in Sec. 1.3 and later (an index is provided at the end of the dissertation).

VLIW	**Superscalar Out-of-Order Approach**
Packs multiple fixed-length operations into a very long instruction word	Variable-length encoding of instructions, no encompassing structure
Static scheduling	Dynamic scheduling
No register renaming in hardware	Register renaming to remove false dependences
Static resource binding	Dynamic resource binding
Limited hazard detection	Processor automatically stalls on hazards
Interwoven instruction-set architecture (ISA) and microarchitecture	Clear separation between ISA and microarchitecture

Table 1.1: Typical characteristics of VLIW and superscalar designs: Those adopted by IPF are provided with a grey shadow.

The Itanium Processor Family—also known as IA-64—is designed as a synthesis of superscalar and VLIW design principles [HP03]: it tries to combine the flexibility and scalability of superscalar architectures—the prevailing standard in desktop and server computing—with the simplicity and efficiency of VLIW, mostly known from embedded systems. Table 1.1 shows which basic characteristics it adopts from both worlds: it borrows from VLIW the concept of instruction words—here called *bundles*—that contain several slots with fixed-length instructions. In contrast to classic VLIW designs, however, the structure of the words is more flexible: there is no strict correspondence between the slots and the processor's execution units, instead the compiler selects between different bundle templates that define this correspondence (detailed later in Sec. 2.1.1.2). This helps combat a VLIW-specific problem, namely the code size increase due to nops (empty instruction slots).

The flexible bundling scheme also aids in ensuring binary compatibility between different implementations (microarchitectures). These implementations are intended to be simple in-order designs that rely on static instruction scheduling through the compiler to achieve sustainably high instructions-per-clock (IPC) rates. The IPC rate is one of the determining basic factors of execution speed, as expressed by the following "iron law" of processor performance [RML+01]:

$$\text{Performance} = \frac{\text{IPC} \times \text{Frequency}}{\text{Instruction Count}}$$

The compiler extracts parallelism and communicates it to the hardware by marking groups of instructions as independent (*EPIC* - "Explicitly Parallel Instruction Set Computing"), thereby guaranteeing that no control and data dependences obstruct their execution in parallel. There is no need for complex, power-consuming circuits that detect and schedule parallelism among instructions (dynamic out-of-order execution). On modern out-of-order processors, the flexibility of dynamic scheduling is restricted anyway by their long pipelines: The scheduling of the instructions often occurs several pipeline stages before the actual execution [HSU+01, TDF+02]. When scheduling a load, this has the consequence that the hardware scheduler does not know which latency this instruction will later experience. Thus, when scheduling dependent instructions, it must speculatively assume an L1 cache hit to benefit from the best-case latency. If the load misses the L1 cache, however, then all scheduled dependent instructions have to be nullified and issued again ("replayed" [HSU+01, TDF+02]).

Another limitation of hardware schedulers is imposed by the scheduling scope. Though the instruction windows of state-of-the-art RISC processors can hold a hundred or more instructions, the sizes of the actual *scheduling windows* are much smaller. On the IBM POWER4 microprocessor, for example, a total of 100 instructions can be active throughout the out-of-order segment of the pipeline, but the scheduler (issue queue) for each integer unit can choose only between at most 18 instructions [TDF+02] (on the AMD Opteron, the ratio is 8 out of 72 [Adv04]). Moreover, dynamic schedulers usually only view instructions from *one* predicted program path—they cannot process speculatively individual critical-path instructions from different control flow paths.

In contrast, a static scheduler overlooks virtually the whole program, enabling exact knowledge of critical data dependence paths within multiple program paths. It also has the time to base scheduling decisions on more thorough analyses (milliseconds instead of nanoseconds). On the downside, however, it suffers from other, inherent limitations of the scheduling scope, such as those imposed by procedure and loop boundaries.

Out-of-order execution is often combined with dynamic register renaming to remove false data dependences (see Sec. 1.3.2). In EPIC, this task is delegated to the compiler, too, which can utilize a vast number of architecturally visible registers to prevent false register dependences (128 general purpose registers alone). As with static scheduling, this does not preclude register renaming in hardware, but it relieves the need for it.

The order of instructions inside the bundles determines—in combination with the bundle templates—deterministically the allocation of instructions to execution units (*resource binding*). Processor-specific *dispersal rules* describe how the issuing takes place. Typically, the code generator optimizes the code for a certain IA-64 microarchitecture and can, by taking these rules into account, *anticipate* and *steer* the resource binding on this processor precisely in order to fully exploit its execution units. When the code is executed on a different implementation with different dispersal rules, then the dispersal logic automatically adapts the statically encoded resource binding to the actual configuration of execution units for correct execution. Hence the resource allocation at runtime may diverge from the encoded plan to ensure compatibility, but this may come at the cost of a performance degradation.

The resource binding is a typical example of the division of responsibilities between processor and compiler in EPIC: On RISC and CISC architectures, the compiler only specifies *what* has to be computed and leaves the details of the execution to the processor. In contrast, a VLIW compiler also determines *how* the instructions are to be executed, taking detailed pipeline characteristics and latencies into account; in a strict VLIW approach, a mismatch between these assumptions and the actual hardware properties can even affect the correctness. In the EPIC approach, the compiler merely specifies how the code *can* be computed. In other words, it anticipates the execution on a certain target microarchitecture and optimizes the code for this design—executing it on a different implementation may then result in performance losses (larger than comparable losses on RISC architectures), but is guaranteed to never impair the correctness.

This is also the reason why Itanium processors feature—in contrast to some VLIWs—a full detection of pipeline hazards and stall the pipeline automatically until they are resolved: they cannot rely on the assumption that the code generator takes pipeline specifics into account. There is a clear line between the *instruction set architecture (ISA)*, which defines the semantics of the code, and the specification of the *microarchitecture*, which describes the rules of instruction issuing and execution. Each implementation complies with this ISA to ensure full binary code compatibility, but it relies on specifically optimized code to unleash its performance potential. As a result, the performance depends to a large extent on decisions made statically at compile time.

1.1 The Quest for Instruction-Level Parallelism

The main motivation for the use of static scheduling is the perception that it paves the way to an almost unlimited amount of *instruction-level parallelism (ILP)* [RF93], which is regarded as the key to overcoming RISC performance barriers [JH00]. Sustainable IPC rates of more than eight are difficult to realize with an out-of-order design[1], but relatively easy to implement on an EPIC

[1]Peak rates of this magnitude are possible even with today's designs, but the sustainable rates are much lower (3-5 IPC) since they are bound by the lowest throughput of all pipeline stages. Main limiters are instruction fetch and retirement logic [HP03].

processor. A different question is, however, how the compiler can extract such parallelism in the presence of obstructive control and data dependences—or simply, how much *can* be extracted at all.

Various *limit studies* have tried to measure—independently of the processor hardware—the amount of instruction-level parallelism in programs. The results vary naturally with the used compiler and instruction set architecture, but they have in common that they show a large gap between *local* and *global* ILP: locally inside basic blocks, typically not more than 2-3 IPC is measured [RF93]. But under the assumption that perfect branch prediction resolves all control dependences—i.e., the whole execution trace is regarded as a single basic block—an average of more than 20 IPC could be measured on the SPECint95 benchmark [PGTM99]. This underpins the importance of moving instructions globally between basic blocks to increase the parallelism (*global scheduling*).

A further study that is based on an Itanium compiler even measures an IPC of more than 30 in the instruction stream of the same benchmark [LWT00]. However, it relies on perfect microarchitecture assumptions, namely

- perfect caches with a single-cycle access time,
- perfect branch prediction that resolves all control dependences,
- the removal of all false dependences via renaming, and
- an unlimited scheduling window.

However, when the scheduling window is constrained to function and loop boundaries—which are both realistic assumptions for static schedulers—the average IPC drops to 12 and 8, respectively. This is still well above the 6 IPC the Itanium 2 is capable to execute; moreover, it can be assumed that there is still room for ILP-increasing improvements in high-level optimizations and code selection. However, since in practice the parallelism is distributed unevenly in the schedule due to control and data dependences, it is inevitable that in many cycles some of the six execution slots are unused.

The EPIC philosophy proposes to use such empty execution slots for *speculation*: the early, tentative execution of code even if it is not yet known if the result will be *needed and correct* at a later point of time. If this is not the case, the result is discarded, otherwise it is available earlier, according to the motto: “Nothing is faster than something that is already done!” [Tri00]. In a simplified view, the potential benefit of speculation is better than leaving the slots unused (in practice, the cost-benefit calculation is often more complex). Compiler-controlled speculation can be regarded as the second pillar of the architecture behind ILP; it is supported by dedicated speculative loads (see Sec. 2.1.5). It helps utilize the full scale of parallel execution units (the *width* of the machine) even on programs with little inherent parallelism.

The reality, however, paints a rather sobering picture so far. Studies of IA-64 code generated by current compilers revealed an average static IPC of 2.5, which is far behind what the hardware could process and what the limit studies promise. We will take a close look at these numbers in Sec. 2.3—there are two possible explanations for them: Either the limit studies are too simplistic so that their results are misleading, or it is indeed possible to expose the predicted ILP in a schedule, but the employed scheduling heuristics fail to do so. As described in detail in Sec. 3.3.1, these algorithms have to deal with strongly interdependent scheduling decisions which involve complex trade-offs. Moreover, instruction scheduling is—even when restricted to basic blocks—an $\mathcal{NP}$-complete problem for which heuristics deliver only approximations [GJ79]. On the whole, the result is a "suboptimal combination of suboptimal partial results" [Käs00a]. The question of the unexploited scheduling potential could only be clearly answered by computing *globally optimal schedules*.

Apart from code generation difficulties, the Itanium Processor Family is further challenged by two powerful trends that have emerged and aggravated during its decade-long development: the *memory wall* and the *thermal wall*. The former gained attention in the middle of the nineties when it became apparent that the memory latency does not scale with the same speed as the processor frequency, leading to ever-growing access times in terms of processor cycles [WM95].

Powerful cache hierarchies have been successfully used to alleviate this gap, but for fundamental reasons like wire delay, the latencies of caches are also growing with their sizes [RML$^+$01, SR03]. They can range from one cycle for the first-level cache to a dozen and more for an on-chip L3 cache (see Sec. 2.2.2). This hard-to-predict variance in the latencies adds a lot of dynamics to the execution process. It can be tolerated better by out-of-order designs, which can rearrange the schedule on a cache miss at runtime (although with limitations, as described above). In contrast, an EPIC processor is typically an in-order design, which must follow the static schedule—if an instruction needs a value that the cache has not yet delivered, the processor stalls. At least, the scheduler can try to minimize these stalls via prefetching and by moving loads as far away as possible from their uses (see Sec. 6.5).

The second barrier is imposed by a paradigm shift in microprocessor design driven by power consumption: classical scaling, which allowed processor designers over decades to integrate ever more complex structures onto the chip while clocking them at the same time at ever higher frequencies, is hitting a "thermal wall". As the heat dissipation reaches critical levels in current manufacturing processes, power is becoming the major limiting factor in processor design [RML$^+$01, HP03]. The resulting constraints on the complexity of the microarchitecture are in line with the "simple hardware promise" of EPIC, but contradict the idea of abundant execution resources that are lavishly used for speculation. To deal with power constraints, future Itanium processors might be more restricted, making code generation for them even more difficult.

This thesis applies *integer linear programming (ILP)*[2] to subphases of the code generation problem on this architecture in order to obtain *globally optimal* and *provably correct* solutions. ILP is a proven combinatorial optimization method that has a long tradition in modeling scheduling problems, also in the area of code generation (Sec. 8.2.2 gives a survey). As a search-based, exact approach it has the ability to *resolve all interdependences* between scheduling decisions and to find in the space of all possible solutions one that is optimal under a given objective function.

Obtaining optimal solutions for $\mathcal{NP}$-complete problems may be computationally demanding—in our case it can take up to a few minutes for individual routines. Hence this approach

[2]From now on, "ILP" is meant to refer to this term instead of "instruction-level parallelism".

is—like most earlier ILP-based approaches to code generation—not suited for today's product compilers. Instead, it is intended for use in professional optimization tools, or in research to explore the potential of EPIC architectures. In contrast to all heuristics and limit studies, it provides theoretically funded insights into the chances and limits of this design direction by delivering concretely the best achievable solutions.

1.2 Overview of this Thesis

After an introduction to frequently used notation and the basics of instruction scheduling in the next section, **Chapter 2** presents the Itanium architecture in detail. It contains an overview of the instruction set architecture, followed by a detailed description of the Itanium 2 microarchitecture, which is the optimization target of this work. The chapter concludes with an analysis of the current status of IPF and the remaining challenges.

Chapter 3 gives an overview of code generation and basic program representations in general. It then provides a survey of existing global instruction scheduling heuristics. The common ground of these algorithms serves as the basis for the development of a unifying, formal definition of the global scheduling problem. Several subproblems are defined, too, in which certain constraint classes are ignored.

Chapter 4 contains a brief introduction to integer linear programming with a special emphasis on solution efficiency. Well-structured formulations are characterized, which are crucial to moderate solution times. On the basis of these insights the main ILP model is then developed in **Chapter 5**. To simplify this process, the previously defined subproblems are dealt with separately. The modeling is accompanied by formal proofs of correctness and well-structuredness. Two $\mathcal{NP}$-completeness proofs are encountered that reveal the factors that contribute to the hardness of the problem. The chapter concludes with a refinement and a summary of the unified model.

Chapter 6 incorporates further variants of code motion and speculation into the model, some of them involving Itanium-specific features. The goal of this chapter is to identify profitable extensions that have the potential to reduce the schedule length further and to integrate them with a minimal complexity increase into the model. **Chapter 7** then describes our implementation of the ILP method as a postpass optimizer. It focuses on those parts that contribute to further improvements in solvability, like precomputations that reduce the search space, or optimizations during the constraint generation. Then it presents the experimental setup and the results.

After an overview of related work in **Chapter 8**, the thesis concludes with a summary and an outlook in **Chapter 9**. The appendix contains several proofs that are not central or too extensive to be included in the main part. A **list of symbols** as well as an **index** are also given at the end of the dissertation.

1.3 Fundamentals and Basic Notions

1.3.1 Graphs and Paths

We employ several types of directed acyclic graphs (*acyclic digraphs*) throughout the thesis and use the following notation: Given a digraph $G = (V, E)$, we denote by $G[V']$ for any $V' \subseteq V$ the subgraph of G induced by this node subset. *Paths* are given by sequences of nodes or edges. The *inner nodes* on a path are all traversed nodes except the start and the end node. A node is called a *predecessor* of another node in the graph if a nonempty path goes from the former to the latter; it is called *direct predecessor* if this path consists of exactly one edge. The definition of *successor* is analogous. $V^{\prec}(x) \subseteq V$ and $V^{\prec}(V') := \bigcup_{x \in V'} V^{\prec}(x)$ denote all predecessors of a given node $x \in V$ and a subset of nodes $V' \subseteq V$, respectively. We define $V^{\preceq}(x) := V^{\prec}(x) \cup \{x\}$ and $V^{\preceq}(V') := V^{\prec}(V') \cup V'$, as well as $V^{\succ}$ and $V^{\succeq}$ analogously.

We call a path *maximal* or *complete* if it starts at a node without predecessor and ends at a node without successor. In the context of a graph, $\mathcal{C}$ refers to the set of all complete paths and $\mathcal{C}(x) \subseteq \mathcal{C}$ and $\mathcal{C}(V') := \bigcup_{x \in V'} \mathcal{C}(x)$ to those complete paths only that pass through a given node $x \in V$ and a subset $V' \subseteq V$, respectively. $\mathcal{C}^{\succeq}(x)$ denotes the set of those paths that start at $x \in V$ and end at a node without successor.

1.3.2 Instruction Scheduling

Instruction scheduling reorders instructions with the typical primary objective to minimize the schedule length subject to several scheduling constraints. The latter guarantee that the resulting schedule is *feasible*, i.e., that it complies with the rules of the target instruction set architecture (ISA), and that the semantics of the program is preserved.

Preserving semantics requires preserving *data dependences* [WM97, HP03, SS02]. Two instructions are *data dependent* if they read or write the same components of the machine state in such a way that reordering them would change the outcome of the computation. The machine state directly read or written by instructions—also called *storage resources* or simply *resources*—usually consists of processor registers and memory locations; data dependences due to these two storage resource classes are termed *register* and *memory dependences*, respectively. The resources read by an instruction are called its *input* or *source operands*, the written ones its *output* or *destination operands*.

An instruction is termed a *use* (*definition*) of a storage resource if it reads (writes) it. A definition is said to *reach* a *program point*[3] if there is a path in the program from the definition to the point that contains no other definition of the resource. These notions are typically used with respect to *(processor) registers* or, more general, *variables* or *temporaries*. The definitions that reach a given use are termed its *reaching definitions*. If the set of reaching definitions contains two or more instructions, then these instructions are termed *concurrent definitions*.

[3]The term program point refers typically to a point between two adjacent instructions. Sometimes we use "instruction" synonymously with "program point". Then it depends on the context if the program point immediately before or after the instruction is meant.

A variable *reaches* all program points that are reached by a definition of the variable. It is said to be *live* at all program points from where there exists a path to a use that contains no other definition of the resource—informally, these are the points where the variable is "in use" and will (possibly) be read at a later point of time. The intersection of both sets of program points—where the variable is both reaching and live—is termed the variable's *live range*. This range starts at definitions of the variable and ends at its *last uses*, which are the uses from where no further uses, but possibly redefinitions can be reached.

Data dependences can be classified into three categories: From an instruction m to a consecutive instruction n there exists[4] a

- *RAW (read-after-write)* or *true dependence* if n reads a resource that is written by m. RAW dependences describe the data flow between instructions, i.e., the operation performed by n needs the result of instruction m as input. In this context n is called *consumer* and m *producer*. These *flow dependences* are inherent to the program semantics and can, in contrast to the next two classes, not be overcome.

- *WAR (write-after-read)* or *anti dependence* if n writes a resource that is read by m. Then n must not be executed before m since it would otherwise overwrite the value to be read by m, changing the program semantics.

- *WAW (write-after-write)* or *output dependence* if n and m write the same resource. Then reordering the two instruction would have the effect that m overwrites the value written by n instead of the other way round, resulting in a different machine state.

In contrast to true dependences, the latter two classes are *name dependences*, which arise if two instructions happen to use the same resources although there is no data flow between them; therefore they are also called *false dependences*. During register allocation, the code generator can prevent the emergence of false register dependences between two instructions by allocating different registers for them. Fig. 1.1 demonstrates this on the basis of a small code sequence.

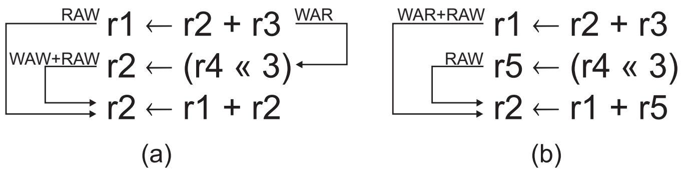


Figure 1.1: Example: The two false register dependences in (a) can be removed by renaming two references to register r2 (b).

In practice, the limited and often small number of *architected registers* (i.e., those exposed by the instruction set) makes false dependences in the assembler code inevitable. If the microprocessor has a greater number of physical registers, however, these dependences can be removed

[4]We consider here only these two instructions in an isolated way; the presence of other instructions can change the data dependences. This is allowed for in the definition of the global data dependence graph in Sec. 3.2.

internally by mapping the architected registers onto the larger number of physical registers (*register renaming*) [HP03]. This is performed by many modern RISC processors, but not on the Itanium with its 128 architected registers (see Sec. 2.2.3.3).

Data dependences are often described by an acyclic digraph with the instructions as nodes. An edge (m, n) then states that n is dependent on m. There can be an integer w_{mn} associated with each edge, which means that n must be scheduled at least w_{mn} cycles after m. These hardware-specific *latencies* are detailed for the Itanium 2 in Sec. 2.2.3.3. In Sec. 3.2 we also give a more precise definition of the data dependence graph.

Using this notation, we can formulate a simple (but nevertheless $\mathcal{NP}$-complete [GJ79]) scheduling problem, namely *local scheduling* for *basic blocks* (regions of straight-line code without control flow, with a single point of entry and a single point of exit). A basic block schedule can be regarded as a timetable that lists the instructions to be executed in consecutive clock cycles.

The number of instructions that can be scheduled per cycle is naturally limited by factors like the processor's number of *execution units* (or synonymously, *functional units*). The instruction scheduler ensures that the target processor can follow the schedule, i.e., that it is in fact capable of executing the instructions scheduled at a cycle in parallel. A simple machine model assumes that each instruction can execute on one or more *execution unit types*. The processor may have several instances of each type. For example, we can assume that a processor has two ALU units (instances) and one load unit, and that an addition operation can be executed on both the ALU and load unit types (while a load can only be issued to the load unit type).

Formally, a schedule can be described by a mapping of instructions to tuples of cycles and execution unit types:

Definition 1.3.1 (Local Instruction Scheduling) Local instruction scheduling is the following minimization problem: Let a data dependence graph $G_D = (V, E_D)$ be given. Furthermore, let R denote the set of all execution unit types of the target processor, R_k the number of available instances of each unit type $k \in R$, and $R(n) \subseteq R$ the subset of those unit types where an instruction n can be executed.

The scheduling problem is to find a *schedule* $\sigma : V \longrightarrow \{1, \ldots, T\} \times R$ with minimal T that satisfies $(\sigma(n))_2 \in R(n)$ for all $n \in V$ plus the following two constraints[5] :

$$\forall 1 \leq i \leq T, \forall k \in R : |\{n \in V \,|\sigma(n) = (i, k)\}| \leq R_k \tag{1.3.1}$$

$$(m, n) \in E_D \Longrightarrow (\sigma(m))_1 + w_{mn} \leq (\sigma(n))_1 \tag{1.3.2}$$

The constraints (1.3.1) and (1.3.2) are called *resource* and *precedence constraints*, respectively.$\square$

Equ. (1.3.2) implies that the length of the longest path in the data dependence graph is a lower bound on T (if the length of each edge (m, n) is its associated latency w_{mn}). Such a maximal-length path is called a *critical path*.

[5] $(\ldots)_i$ denotes the i-th component of the tuple.

Chapter 2

The Itanium Processor Family

As mentioned before in the introduction, the Itanium Processor Architecture strictly separates between the instruction set architecture (ISA) and the microarchitecture to ensure compatibility between implementations. We will adhere to this separation in this chapter and present first the basic concepts and features of the ISA. The second part then describes in detail their implementation on the Itanium 2 processor, the second-generation design that is targeted by this work.

2.1 The IA-64 Architecture

The IA-64 instruction set architecture comprises all the information that is necessary to write programs that execute semantically correct on any Itanium processor. This information is available in three volumes of the manufacturer's "Software Developer's Manual": The first covers the application architecture [Int02a] (programming environment, optimization, etc.), the second the system architecture [Int02b] (virtual memory, interrupt model, etc.), and the third provides a comprehensive instruction set reference [Int02c]. This section focuses on the application architecture. The concepts are presented on a rather abstract level first and then illustrated by an example.

2.1.1 Fundamentals

2.1.1.1 Execution Unit and Instruction Types

The Itanium architecture categorizes execution units and instructions into different types. There are four execution unit types:

- **M-Unit:** Memory Unit
- **I-Unit:** Integer Unit
- **F-Unit:** Floating-Point Unit
- **B-Unit:** Branch Unit

Five instruction types are distinguished:

- **A-Type:** Common ALU instructions (simple arithmetic, boolean operations)
- **I-Type:** More special integer computation instructions like shifts
- **M-Type:** Memory instructions like loads and stores
- **F-Type:** Floating-point instructions
- **B-Type:** Branches

Fig. 2.1 shows on which execution units instructions can be executed, depending on the respective types.

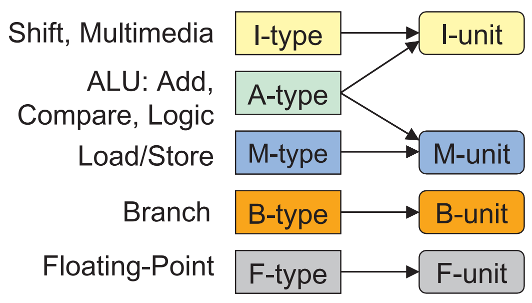


Figure 2.1: Mapping of instruction types to execution unit types.

While the frequent A-type instructions can be executed on both the M- and I-units, the other types can only be issued to their corresponding unit types.

2.1.1.2 Instruction Bundles

Each instruction has a fixed length of 41 bits. Three instructions at a time are grouped together into 128-bit sized and aligned simple structures called *instruction bundles*. Each instruction occupies one of the three *slots* of a bundle (see Fig. 2.2).

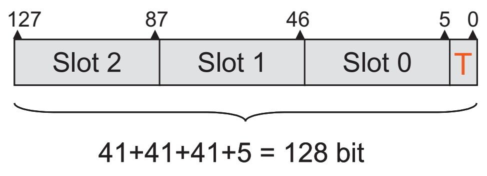


Figure 2.2: Instruction bundle.

The remaining 5 *template bits* in the bundle determine the mapping of the instructions to execution unit types. They specify for each slot the *slot type*, i.e., the type of a unit where the instruction in this slot can be executed. This helps decode instructions and route them early to proper issue ports. Evidently five bits are not sufficient to encode all possible mappings, instead they only select one of the following predefined *bundle templates* (given in the order slot 0/1/2):

$$\mathsf{MII, MMI, MFI, MIB, MMB, MFB, MMF, MBB, BBB, MLX} \tag{2.1.1}$$

A-type instructions can be placed in either an M-slot or an I-slot. The last template is a special case where the last two slots LX contain the double-sized instruction „`movl`“, which loads an encoded 64-bit constant into a register.

The order of instructions, as it is given by increasing bundle addresses and slot numbers, is significant: IA-64 features a strictly sequential execution semantics, i.e., the semantics of a bundle sequence is defined as if the instructions were executed one after the other in the given order. However, as this architecture embraces instruction-level parallelism, sequential execution inside the processor is not intended. Therefore the compiler *marks explicitly* which instructions can be executed in parallel.

For this, the template bits specify besides the unit types also the locations of *stops*. These can be inserted after the slots of the bundle and delimit groups of instructions that are executable in parallel, called *instruction groups*. An instruction group (often only called "group") starts at a branch label or a stop and ends at the next following stop. It can be arbitrarily large and span several bundles—there is no direct correlation between bundle and instruction group boundaries. Figure 2.3 gives an example where the positions of stops are denoted by underscores.

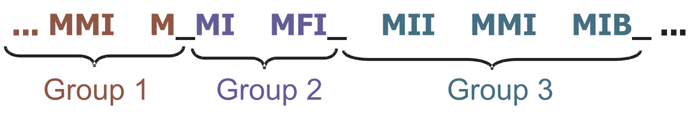


Figure 2.3: Example for instruction groups.

If at runtime the parallel execution of a group as a whole is impossible due to limited execution resources, the group can be split up into several parts that are executed consecutively. Accordingly, the instruction groups encoded in the program are often referred to as *static*, and those occurring at runtime as *dynamic*.

Within a group, RAW (read-after-write) and WAW (write-after-write) register dependences are not allowed; this concerns all types of registers. If any of these restrictions are not met, the behavior of the program is undefined. However, WAR (write-after-read) register dependences are allowed, as are memory dependences, i.e., dependences between loads and stores that can occur at runtime if their accessed memory regions overlap. In this case, the hardware implementation guarantees that these instructions behave as if they were executed consecutively in the given order. However, this may be accompanied by performance penalties (see Sec. 2.2.3). We call dependences that are allowed inside an instruction group *intra-group dependences* and the others *inter-group dependences*.

The placement of stops is significantly restricted and depends on the bundle template. Possible locations of a stop are:

- after the last slot of any bundle (independent of the template), often denoted by an underscore appended to the template, and additionally/or
- between slot 0 and slot 1 of bundle template MMI, written as M_MI, or
- between slot 1 and slot 2 of bundle template MII, written as MI_I.

These architectural restrictions, especially the limitation of intra-bundle stops to the templates MMI and MII, make the process of choosing a compact bundle sequence for a series of instructions (called *bundling*) a challenging task. Empty slots for which no instruction is left must be filled with *nops*, dummy instructions that waste the capacity of the instruction cache. Earlier work by the author has shown that optimal bundling is feasible, using an approach based on dynamic programming and precomputed data [Win01, KW01].

2.1.1.3 Architected Registers and the Register Stack

The IA-64 ISA provides the following architected registers:

- 128 general purpose 64-bit integer registers `r0`-`r127`;
  `r0` always reads as 0, writing to this register is disallowed.
- 128 floating-point registers of 82 bits size;
  `f0` always reads as 0.0 and `f1` as 1.0; writing to these registers is disallowed.
- 64 single-bit predicate registers `p0`-`p63` used in predication and conditional branching;
  `p0` always reads as 1, writing to this register has no effect.
- 8 branch registers `b0`-`b7`, used to specify the target addresses of indirect branches.
- Many application and special registers, some of which are presented below.

IA-64 features *stacked registers* to save and pass register values when a procedure is called. For this purpose, the general register file is divided into a *static* and a dynamically renamed *stacked subset*. The static part (`r0`-`r31`) is equally visible to all procedures; [Int01a] describes conventions of its usage, for example, the global data pointer, `gp`, and the memory stack pointer, `sp`, are usually stored in `r1` and `r12`, respectively.

The stacked subset is local to each procedure and consists of up to 96 registers, always starting at `r32`. As depicted in Fig. 2.4, these registers—also called the *register stack frame*—are further partitioned into two variable-size areas: the local area and the output area. In the figure, the sizes of the total frame (*sof*) and the local area (*sol*) are 20 and 16, respectively. The current configuration of the register stack frame is stored in a special register, the *current frame marker* (*cfm*).

Upon procedure entry, a processor unit called *register stack engine* (RSE) renames the physical registers in such a way that the output area of the caller becomes the new register stack frame of the callee, i.e., the latter obtains its parameters starting from the logical register `r32`. In the

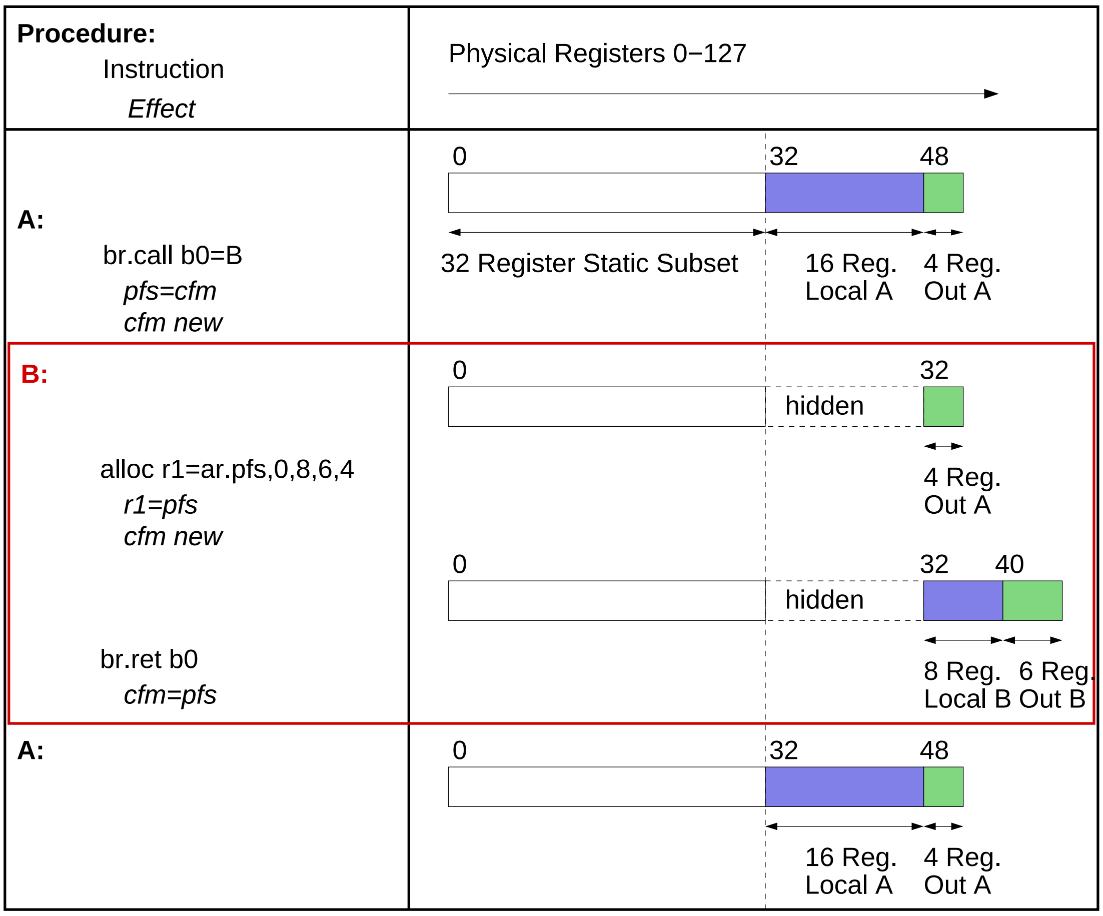


Figure 2.4: Parameter passing through the register stack.
A calls B, the callee B allocates a new register stack frame and returns with `br.ret`. The logical register numbers are given at the bars.

example, the stacked registers are renamed with offset -16. The renaming makes it necessary to differentiate between *logical* or *architected registers* as they are visible to the software, and the *physical* registers inside the processor.

The local registers of the caller are not accessible within the new register stack frame. Before renaming the register areas, the function call `br.call` copies the *cfm* of the caller to the *previous function state* (*pfs*) register. At the end of the routine, the return `br.ret` automatically restores *cfm* from *pfs* in order to reinstall the previous register stack frame.

The callee can increase the size of its register stack frame with the `alloc r1=ar.pfs,i, l,o,r` instruction, which must appear first in an instruction group. The resulting new register stack frame size *sof* is the sum of the specified local area size `i+l` and the size of the output area `o` (in the example 14, 8, and 6, respectively). `alloc` also saves the current *pfs* in the destination register (here `r1`). This is useful if the callee calls subroutines itself: Then this register must be preserved beforehand and written back to *pfs* before executing `br.ret` (using `mov ar.pfs=r1`, not shown in the figure). In the same way, the return address in `b0` must be saved if further subroutines are called.

Parameter passing via stacked registers comes at little cost since the register renaming can be performed in a separate pipeline stage and does not add to the critical path of the machine (see Sec. 2.2.3.3, [JH00]). However, if the depth of the procedure call stack (that is, the hidden registers) exceeds the capacity of the physical register file, parts of it must be spilled to memory or, more precisely, to the L1D cache. This occurs automatically in the background and mostly transparent to application software: The RSE injects spills and fills into unused, idle memory ports for this purpose. Details of this process are described in [Int01a, Int02a, Int02b].

2.1.1.4 Predication

Predication is a means of conditionally executing instructions. The encoding of each instruction contains a 6-bit predicate field that refers to one of the 64 predicate registers, called the *qualifying predicate* (`qp`). If this predicate is true at runtime (i.e., the predicate register has value one), then the instruction is executed as normal. Otherwise it is ignored and has—like a nop—no effect on the program semantics (with one exception that is described below).

Predicate registers are written by compare instructions; they replace the condition flags known from other architectures. Almost every instruction can be predicated, including branches (this is how conditional branches are implemented).

The most significant benefit of predication is its ability to eliminate branches and the associated performance costs like branch misprediction penalties (explained in Sec. 2.2.3) [SW04]. As it replaces control with data dependences, it increases the flexibility during instruction scheduling and thereby the possibilities to extract instruction-level parallelism. In the ideal case, this means merging a complex acyclic control flow structure into a contiguous, branchless code block with a high amount of parallelism, which can be fetched and executed fast without disruptions of the pipeline. This optimization is known as *if-conversion* [SS02].

2.1.2 IA-64 Programming

In general, the syntax of IA-64 instructions is as follows [Int01b]:

```
[(qp)] mnemonic[.completer]* dests=sources
```

- `qp` gives the qualifying predicate; if omitted, `p0` is encoded (which is always one).
- `mnemonic` describes the operation to be executed. Its semantics can be supplemented by appended suffixes, the so-called *completers*.
- `dests` lists, separated by commas, the destination operands. Except for stores, these are always destination registers.
- `sources` lists, separated by commas, the source operands. Except for loads, these are always source registers or constants.

IA-64 is a load/store architecture where memory accesses can only be performed with dedicated load/store instructions. All other instructions have registers as operands. There are architectural limits on the numbers of source and destination operands: Each instruction can read and write at most two general purpose registers, respectively. The same limitation holds for predicate registers. Floating-point instructions can read at most three floating-point registers and write one. Floating-point instructions (F-type) and branch instructions (B-type) cannot access the general purpose register file directly; instead, there are special instructions to transfer values to and from floating-point and branch registers.

The bundling can be left to the assembler, or it can be explicitly stated in the assembly code using the following syntax:

```
{ .TYPE
     Instruction 1  [;;]
     Instruction 2  [;;]
     Instruction 3  [;;]
}
```

`TYPE` denotes the bundle template (2.1.1) without the stops (underscores); the latter are represented by appended double semi-colons after the instructions.

2.1.2.1 Example

Algorithm 1 shows an example routine in pseudo code. The routine recursively sifts down an element in a binary heap until the heap property is satisfied [Hag97]. Its functioning is secondary, we will just use it to demonstrate the semantics of IA-64 assembler on the basis of a hand-coded translation (shown in Alg. 2). The lines of the translation are labeled with characters. In the following, we will refer to the instruction in line “`A`” simply as “instruction `A`” etc.

Algorithm 1 Example routine SiftDown: Moves `A[i]` downwards in a binary heap `A[1...n]`.

```
SiftDown(i,n,A):
 Label:
  k := 2i;
  if (k > n) return;
  if (k+1 <= n) {
     if (A[k+1] < A[k]) k++;
  }
  if (A[i] > A[k]) {
     Swap A[i] and A[k];
     i := k;
  } else return;
  goto Label;
```

The first instruction `alloc` manages the register stack. It reserves the eight registers `r32`-`r40` for use inside the routine. As described before in Sec. 2.1.1.3, the parameter passing during a subroutine call takes place via renaming of register areas. After the renaming, the first parameter—the address of the first heap element here—is to be found in `r32`. It is saved to `r34` by instruction `B`. For this, an add instruction with second operand `r0`$= 0$ is used (often written as the pseudo-op `mov r34 = r32`).

In `D`, the first heap element is loaded into `r39` from the memory address in `r34`. The number in the mnemonic of the load specifies the size of the value to be loaded (1, 2, 4 oder 8 bytes). For sizes less than eight bytes, the loaded value is zero-extended to 64 bits. Since there is a RAW dependence from instruction `B` to `D`, they must be separated by a stop (after the first bundle).

The compare instruction `cmp` with completer `gt` in `E` tests if the value in `r35` is greater than that in `r33`. The result and its complement are written into the first and second destination predicate register `p1` and `p2`, respectively. If the condition is true, the subroutine should return. This is performed by the branch instruction `F` with completer `ret`, which is executed if and only if its qualifying predicate is true. Its argument is the return address, which has been saved in the branch register `b0`.

We note that the compare and the branch are in the same instruction group although they are RAW dependent with respect to `p1`. This is an exception that is only allowed if the consumer of the compare is a branch. Sec. 2.2.3.3 will show that this is possible because branches are resolved in a later pipeline stage than the computation of compares.

The instructions `D`-`H` are independent and constitute an instruction group—their execution can commence in parallel in the processor pipeline. However, if the branch `F` is taken, any available results of `G`-`H` must be discarded. Hence, at runtime, instruction groups can end prior to a stop, namely at the first taken branch. The next group then commences at the bundle pointed to by the branch target address[1].

[1]Branch targets are not individual instructions, but 128-bit aligned bundle addresses. The execution always resumes at the first slot of the bundle. This simplifies the hardware, but has the drawback that each basic block

Algorithm 2 Assembly translation of Algorithm 1 (with 64-bit integers as heap elements).

```
SiftDown:
 {   .mii
A       alloc r11 = ar.pfs, 0, 8, 0, 0
B       add r34 = r32, r0                //r34=&A[i]
C       add r35 = r32, r32  ;;           //r35=&A[k]    (2)
 }
Label:
 {   .mib
D       ld8 r39 = [r34]                  //r39=A[i]
E       cmp.gt p1, p2 = r35, r33         //r33=n        (3)
F (p1)  br.ret b0
 } { .mii
G       ld8 r37 = [r35]                  //r37=A[k]
H       add r36 = 8, r35  ;;             //r36=&A[k+1]
I       cmp.le p1, p2 = r36, r33  ;;     //             (4)
 } { .mmi
J (p1)  ld8 r38 = [r36]  ;;              //r38=A[k+1]
K (p1)  cmp.lt.unc p3, p4 = r38, r37     //             (5)
L       nop.i   0  ;;
 } { .mmi
M       nop.m   0
N (p3)  add r35 = 8, r35                 //r35=&A[++k]  (5)
O (p3)  add r37 = r38, r0  ;;            //r37=A[++k]   (5)
 } { .mib
P       cmp.gt p1, p2 = r39, r37         //             (7)
Q       nop.i   0
R (p2)  br.ret b0                        //             (10)
 } { .mib
S       st8 [r34] = r37                  //             (8)
T       add r34 = r35, r0                //             (9)
U       nop.b   0
 } { .mib
V       st8 [r35] = r39                  //             (8)
W       add r35 = r35, r35               //        (9), (2)
X       br Label
 }
```

The if-clause from lines 4-6 of the source program is translated via predication. The condition of the if-clause is computed into `p1` and the instructions `J` and `K` are predicated with this register. The instruction `K` computes into `p3` the condition of the inner if-clause of line 5. The instructions `N` and `O` of this inner if-clause then should only be executed if both `p1` *and* `p3` are equal to one, i.e., if both conditions of the nested if-clause are true.

However, instructions can only be guarded by one predicate register (6 bits in the encoding must be sufficient). The following approach, for example, does *not* compute the nested conditional properly in every case:

```
K  (p1)  cmp.lt p3, p4 = r38, r37   ;;  //
N  (p3)  add r35 = 8, r35               //inner clause
O  (p3)  add r37 = r38, r0              //
```

The problem with this sequence is that the compare does not write `p3` at all if its own qualifying predicate is zero. Then the value of `p3` is undefined (but should be zero). This could be handled by initializing it to zero beforehand, a plainer solution (used in Alg. 2) uses an *unconditional compare* (with the completer `unc`), which writes both predicate target registers with zero if predicated off. This is remarkable as this instruction modifies architectural state even if its qualifying predicate is false.

Further worth noting is the last instruction group with two intra-group dependences:

- The instructions `T` and `S` are WAR dependent since the store `S` writes the value in `r37` to the memory location addressed by `r34`, and reads for this purpose both registers. Stores are the only instructions that have source registers, namely the address register, denoted on the left-hand side of the equal sign (enclosed in square brackets, which symbolize that the target operand is a memory location).

- Similarly, the instructions `W` and `V` are WAR dependent.

The instructions `L`, `M`, `Q`, and `U` are nops that are unavoidable due to the bundle structure.

2.1.3 Instruction Set Overview

The IA-64 instruction set is large and diversified; only the most common instructions are listed in the following tables with their syntax and semantics. For most instructions, the semantics is described in a C-like pseudo code. Technically similar instructions that execute on the same units with the same latency on Itanium processors are arranged in groups.

Table 2.1 lists **A-type instructions**: integer addition, subtraction, a shift-left-and-add instruction used for address computations, logic operations, and a compare instruction in many variations. All these instructions (except for the logic instructions) exist also in SIMD variants (“multimedia instructions”) that treat the general registers as concatenations of eight 8-bit, four

with n instructions inherently must contain at least n modulo 3 nops. If the basic block sizes are assumed to be distributed randomly, this results in one additional nop per basic block on average.

Group	Syntax	Semantics

IALU	add r1=r2,r3
	add r1=r2,r3,1
	add r1=imm,r3
		r1=r2+r3
		r1=r2+r3+1
		r1=imm+r3

	sub r1=r2,r3
	sub r1=r2,r3,1
	sub r1=imm,r3
		r1=r2-r3
		r1=r2-r3-1
		r1=imm-r3

	shladd r1=r2,imm,r3	r1=(r2<<imm)+r3

ILOG	and r1=r2,r3
	and r1=imm,r3
		r1=r2&r3
		r1=imm&r3

	andcm r1=r2,r3
	andcm r1=imm,r3
		r1=r2&~r3
		r1=imm&~r3

	or r1=r2,r3
	or r1=imm,r3
		r1=r2|r3
		r1=imm|r3

	xor r1=r2,r3
	xor r1=imm,r3
		r1=r2^r3
		r1=imm^r3

ICMP	cmp.*CR*.*CT* p1,p2=r2,r3
	r2 can also be 'imm'
	CR=eq,ne,lt,le,gt,ge
	ltu,leu,gtu,geu
	CT=ε,unc,or,
	and,or.andcm
		p1=(r2 *CR* r3)
		p2=~(r2 *CR* r3)
		CR=!=,=,<,<=,>,>=
		u = unsigned
		See text

MMALU_A	padd*X*.*COMP* r1=r2,r3 *X*=1,2,4 *COMP*=ε,sss,uuu,uus	e1=e2+e3 SIMD element size See manual
	psub*X*.*COMP* r1=r2,r3 *X*=1,2,4 *COMP*=ε,sss,uuu,uus	e1=e2-e3 SIMD element size See manual
	pavg*X*.*COMP* r1=r2,r3 *X*=1,2 *COMP*=ε, raz	e1=(e2+e3+1)>>1 SIMD element size See manual
	pavgsub*X* r1=r2,r3 *X*=1,2	See manual SIMD element size
	pshladd2 r1=r2,c3,r3	e1=(e1<<c3)+e3
	pshradd2 r1=r2,c3,r3	e1=(e1>>c3)+e3
	pcmp*X*.*PR* r1=r2,r3 *X*=1,2 *PR*=eq,gt	e1=(e2 *PR* e3) See manual

Table 2.1: A-type instructions.

16-bit, or two 32-bit elements. They perform the operation on each of these elements (denoted by `e` in the table) independently and in parallel.

In addition, more complex SIMD operations are available as **I-type instructions** (Tab. 2.3): these include parallel multiply, parallel shift and a combination of both, as well as highly specialized parallel minimum and maximum operations, and pack and unpack instructions (which convert between different element sizes). The I-type instructions also comprise several non-SIMD shift instructions that shift the value of a general register by an amount specified by another general register (*variable shift*) or an encoded constant (*fixed shift*). While the variable shifts `shr` and `shl` are technically similar to SIMD instructions, the fixed shifts are actually performed by the more general shift-and-mask instructions `dep` and `extr`, which move bit fields to different bit positions. Further I-type instructions transfer values between different register files.

The **M-type instructions** (Tab. 2.4) include loads, stores, and the prefetch instruction `lfetch`. The latter can be employed by the compiler to move the addressed line to a location in the memory hierarchy in order to speed up future expected accesses to this line. The effect of `lfetch` is comparable to a load without a destination register (there are other implementation-specific differences). The intended destination location inside the memory hierarchy is specified by a *locality hint* (given by a completer): for instance, the `nt1` completer indicates that the data should not be prefetched into the highest level of the cache hierarchy, but to all lower levels. This can be used to prevent the congestion ("pollution") of a small L1 cache if large amounts of data are prefetched. Loads and stores support these locality hints, too. They do not affect the functional behavior of the program, but the performance in an implementation-specific manner.

All memory instructions support post-increment, i.e., an additional source operand that is added to the address register after the memory access. Both immediate and register post-increment are defined for loads and prefetches; stores, however, only allow immediate post-increment (otherwise there would be three source registers).

The `getf` and `setf` instructions are used to transfer integers from and to floating-point registers, respectively.

The **B-type instructions** (Tab. 2.5) comprise IP-relative branches, calls and returns (explained in Sec. 2.1.1.3), and indirect branches, which use branch registers to specify the branch target address. Most of these branches can be made conditional via predication. The compiler can also encode a branch hint, a completer that signals whether the branch should be predicted taken (`dptk`, `sptk`) or not-taken (`dpnt`, `spnt`). While the `dpxx` completers only predefine the branch direction for the cases where dynamic branch prediction information is not yet available, the `spxx` completers indicate that no dynamic prediction resources should be allocated at all for a branch (this can be used to mark branches that are most likely to be not-taken, for example to error handlers).

These branch hints can also be provided earlier in the code by specific *branch predict instructions*, along with information about the location and the target address of the upcoming branch. If scheduled several cycles before the actual branch, this information can be used by the processor to prepare the branch execution, for instance by prefetching instructions from the branch target address into the instruction cache.

The **floating-point (F-type)** (Tab. 2.6) arithmetic instructions support the internal 82-bit floating-point register format as well as single, double or double-extended real formats according

Group	Syntax	Semantics
ISHF	dep r1=r2,r3,p,len	Deposits bit fields
	dep r1=imm,r3,p,len	See manual
	dep.z r1=r2,p,len	Variant with r3=0
	dep.z r1=imm,p,len	See manual
	extr r1=r3,p,len	Extracts bit fields
	extr.u r1=r3,p,len	See manual
FRBR	mov r1=b1	Reads branch registers
TOBR	mov b1=r1	Writes branch registers
FRAR	mov r1=lc	Reads the registers
	mov r1=pfs	lc and pfs
TOAR	mov lc=r1	Writes the registers
	mov ec=r1	lc, ec and pfs
	mov pfs=r1	imm operand also possible
FRPR	mov r1=pr	Reads predicate registers
TOPR	mov pr.rot=imm	Writes predicate registers
CHK_I	chk.s r2,target	Control speculation check
TBIT	tbit.*R*.*CT* p1,p2=r3,p	Tests if bit p in r3
	R=nz,z	is 1 (nz) or 0 (z)
	CT as with cmp	
	tnat.*R*.*CT* p1,p2=r3	Tests NaT bit of r3
MMALU_I	pmax1.u r1=r2,r3	e1=max_unsigd(e2,e3)
	pmax2 r1=r2,r3	e1=max(e2,e3)
	pmin1.u r1=r2,r3	e1=min_unsigd(e2,e3)
	pmin2 r1=r2,r3	e1=min(e2,e3)
MMMUL	pmpy2.r r1=r2,r3	Parallel multiply
	pmpy2.l r1=r2,r3	See manual
	pmpyshr2 r1=r2,r3,c2	Parallel multiply and shift
	pmpyshr2.u r1=...	See manual
MMSHF	pack*X*.sss r1=r2,r3	See manual
	X=2,4	
	unpack*X*.*C* r1=r2,r3	See manual
	X=2,4 *C*=h,l	
	pshr*X*[.u] r1=r2,r3	e1=(e2>>r3) arithmetic
	pshr*X*[.u] r1=r2,imm	e1=(e2>>imm) arithmetic
	X=2,4	u=unsigned
	pshl*X* r1=r2,r3	e1=(e2<<r3)
	pshl*X* r1=r2,imm	e1=(e2<<imm)
	X=2,4	
	shr r1=r2,r3	r1=(r2>>r3) arithmetic
	shr.u r1=r2,r3	r1=(r2>>r3) unsigned
	shl r1=r2,r3	r1=(r2<<r3)
XTD	sxt/zxt/czx r1=r2	Sign extension

Table 2.3: I-type instructions.

Group	Syntax	Semantics
LD	ld*X*.*LDT* r1=[r3] ld*X*.*LDT* r1=[r3],r2 ld*X*.*LDT* r1=[r3],imm *X*=1,2,4,8 *LDT*=ε,s,a,sa,c, c.clr,fill	r1=mem(r3) Post incr. r3 += r2 Post incr. r3 += imm Data size Completers for speculation See text
FLD	ldf*fsz*.*LDT* f1=[r3] *fsz*=s,d,e	f1=mem(r3) Data size
ST	st*X* [r3]=r2 stX [r3]=r2,imm X=1,2,4,8 st8.spill [r3]=...	mem(r3)=r2 Post incr. r3 += imm Data size See manual
LFETCH	lfetch [r3] lfetch [r3],r2 lfetch [r3],imm	Prefetch Post incr. r3 += r2 Post incr. r3 += imm
FRFR	getf.sig r1=f2	r1=f2
TOFR	setf.sig f1=r2	f1=r2
ALLOC	alloc r1=ar.pfs,i,l,o,r	See text

Table 2.4: M-type instructions.

Group	Syntax	Semantics
BR	br.*BT*.*BW* target br.*BT*.*BW* b1=target br.*BT*.*BW* b2 *BT*=cond,call,ret, cloop, ctop, cexit, wtop,wexit *BW*=spnt,sptk, dpnt,dptk	Branch call form indirect form See text Branch Hints See Manual
RSE_B	clrrrb clrrrb.pr	Clear RRB See Manual
BRP	brp.*ipwh*.*ih* target,tag *ipwh* = sptk, loop, exit, dptk *ih*=ε,imp	Branch Predict Branch information See text

Table 2.5: B-type instructions.

to the IEEE standard [HP03]. The table lists only a small selection of all floating-point instructions; many exist also in SIMD variants that treat the register's 64-bit significands as a pair of IEEE single precision values. The result range and precision are determined either statically via the instruction's completer, or dynamically via the precision-control and widest-range-exponent fields in the *floating-point status register* (FPSR).

In the latter case, each instruction refers to one of the four identical *status fields* `sf0`-`sf3` inside FPSR, which serves as a kind of execution context, i.e., which controls and records its execution: The status field specifies the output format and contains IEEE flags that are set according to the result (underflow, overflow, etc.). The purpose of multiple status fields is, for example, that some of them can be used to store the status flags of speculated instructions, which are only later committed to architectural state (which is typically in `sf0`).

Group	Syntax	Semantics
FMAC	fma.*pc*.*sf* f1=f3,f4,f2	f1=f3*f4+f2
	fnma.*pc*.*sf* f1=f3,f4,f2	f1=-(f3*f4)+f2
	pc=.s,.d,none	Precision
	sf=sf0,sf1,sf2,sf3	Status Field

FMISC	frcpa.*sf* f1,p2=f2,f3	f1=f2/f3 ∧ p2=0, or
		f1=approx(1/f3) ∧ p2=1

	frsqrta.*sf* f1,p2=f3	f1=sqrt(f3) ∧ p2=0, or
		f1=approx(sqrt(f3)) ∧ p2=1

	fmax.*sf* f1=f2,f3	f1=max(f2,f3)
	fmin.*sf* f1=f2,f3	f1=min(f2,f3)
FCVT	fcvt.x*uf* f1=f2	Treat 64-bit significand of f2 as an integer, convert to FP
	fcvt.fx*u*.*sf* f1=f2	Convert FP to (*unsig.*) integer
XMA	xma.*xs* f1=f3,f4,f2	Integer mul-add: f1=f3*f4+f2
	x=l,u	f1: lower, upper bits of sum
	s=ε,u	signed, unsigned
FCMP	fcmp.*r*.*t*.*sf* p1,p2=f2,f3	Compare
	r=eq,ne,lt,gt,etc.	like A-type cmp
	t=ε,unc	like A-type cmp

Table 2.6: F-type instructions.

The basic building block of all floating-point computations is the fused multiply-and-add instruction `fma`, which computes a multiplication combined with an addition (a frequent combination in linear algebra). The combined execution of these operations is faster and more precise since only one rounding of the result occurs. If only single additions and multiplications are needed, one of the source registers can be replaced by `f1` ($= 1.0$) and `f0` ($= 0.0$), respectively. Many complex operations like divide, remainder, and transcendental functions are not available in hardware, but explicitly computed by sequences of `fma` instructions [HKST99, CHN99].

Algorithm 3 Sequence to compute `f8=f6/f7` in double precision.

```
A        frcpa.s0 f8,p6 = f6,f7 ;;
B   (p6) fma.s1 f9 = f6,f8,f0
C   (p6) fnma.s1 f10 = f7,f8,f1 ;;
D   (p6) fma.s1 f9 = f10,f9,f9
E   (p6) fma.s1 f11 = f10,f10,f0
F   (p6) fma.s1 f8 = f10,f8,f8 ;;
G   (p6) fma.s1 f9 = f11,f9,f9
H   (p6) fma.s1 f10 = f11, f11, f0
I   (p6) fma.s1 f8 = f11,f8,f8 ;;
J   (p6) fma.d.s1 f9 = f10,f9,f9
K   (p6) fma.s1 f8 = f10,f8,f8 ;;
L   (p6) fnma.d.s1 f6 = f7,f9,f6 ;;
M   (p6) fma.d.s0 f8 = f6,f8,f9
```

For example, Alg. 3 shows a sequence from [Int02a] that computes `f8=f6/f7` in double precision. The first instruction `frcpa` computes an approximation (good to 8 bits) of `1/f7`. Then the remaining instructions perform three (unrolled) iterations of the Newton-Raphson method to compute the correctly rounded value of `f6/f7` [HKST99, MP00]. In special cases, where these iterations are not necessary or sufficient, the result is provided otherwise—either by `frcpa` or by an invoked software handler—then the iterations are predicated off by clearing `p6`.

The routine takes 30 cycles on the Itanium 2 (4 per `fma`, which can be executed on both available F-units). Besides the drawback of code expansion, it has several advantages to compute complex floating-point operations by sequences of simple atomic multiply-and-adds: the hardware is simpler and the FMA units can be optimized and fully pipelined (in contrast to typical hardware implementations of division, square root, etc. [MP00]), boosting throughput and scalability. It is also possible to schedule several sequences in an interleaved manner in order to exploit the parallelism of the pipelined units.

The FMA units on this architecture are also used to compute integer multiplications: The integers are transfered to floating-point registers with `setf`, multiplied with the `xma` command and the result is returned with `getf`.

2.1.4 Multiway Branches

The architecture allows to execute multiple branches in an instruction group in parallel (up to three on the Itanium 2; visible from the bundle templates (2.1.1) in Sec. 2.1.1.2). Then the first branch in the group is taken that is unpredicated or has predicate true. Such *multiway branches* reduce the critical path of blocks with more than two possible successors, which can emerge from transformations like if-conversion (see Sec. 2.1.1.4).

2.1.5 Speculation

In the computer architecture field, *speculation* refers to the early, tentative execution of an operation even if it is not yet known if the result will be *needed and correct* at a later point of time. If the assumptions the speculation was based upon turn out to be true, then the result is available earlier; otherwise, the speculation has failed and the result is discarded.

In the case of successful speculation, its *benefit* can be measured in the number of saved cycles, B, due to the earlier availability of the result. Two kinds of associated costs can be distinguished: a fixed part C_f, which exists regardless of success or failure, and a variable part C_v, which incurs only if the speculation fails. The former can also be *opportunity costs*, i.e., the profit that would have been possible if the resources bound by the speculation had been used otherwise; it is often difficult to quantify this speedup in cycles. The latter can be recovery costs resulting from the roll back of all effects of the speculative action. A speculation is *useful* on average if it has a positive expected benefit

$$pB - (1-p)C_v - C_f$$

where p denotes the probability of successful speculation.

The following two subsections present the two major kinds of explicit speculation featured by the Itanium architecture. Both are directed by the compiler and supported by special hardware. They aim at executing loads earlier by moving them upwards before conditional branches (*control speculation*) and potentially data dependent stores (*data speculation*). This early issuing of loads is generally considered as crucial to cover the memory latency and to avoid load-use stalls.

2.1.5.1 Control Speculation

Control speculation in general denotes the premature execution of an instruction even if it is not yet known whether the execution needs to take place. This can occur by moving code upwards from its original block (termed its *source block*) to a *destination block* that is not postdominated by the former (see later Def. 3.2.5). Then it is executed there earlier at runtime, before it is known that the control flow will reach the source block where its execution is actually *needed*. If the latter is the case, then the result is available earlier and the speculation was successful; otherwise the execution was superfluous and should have no harmful effect on the architectural state.

Most instructions can be executed speculatively (short: *speculated*), i.e., have no harmful side effects if they are executed though this would not have been the case in the original program (see also the detailed discussion in Sec. 6.2). These instructions are often referred to as *speculative*; those without this property are called *non-speculative*. Memory accesses like loads are in general non-speculative since the address used during a superfluous execution could be invalid and trigger a false exception (which would not have occurred in the original program).

Often it can be proven through static analysis that a load is *safe* at a destination block, i.e., that it is never executed with an invalid address there [BRS92]. For the remaining cases, IA-64 supports the deferral of load exceptions: If the control speculative load instruction `ld.s` is used and if the conditions of an exception occur, then the exception is not triggered but instead a special *NaT (Not a Thing) bit* associated with the load destination register is set. This bit signals that the value of the register is void as the load failed due to a suppressed exception. Each of the 128 GPRs has such an additional NaT bit; for floating-point registers, the condition is represented by a special register value, *NaTVal*, which cannot occur as the result of normal computations.

If after the deferral of an exception the source block of the speculated load is not reached, then the set NaT bit has no effect at all—the load destination register is then not live, i.e., it is never read, but will be overwritten (together with the NaT bit) somewhere later.

In the opposite case, if the source block of the load or a control equivalent block is reached, then the exception has been real and must be dealt with. For this purpose a control speculation check instruction `chk.s` is scheduled there that branches to a specified label if its register argument is NaT or NaTVal. At this label the compiler has generated *recovery code* that reexecutes the load—this time in a normal, non-speculative version so that the exception is eventually triggered. After the exception has been handled—and if this has not terminated the program—the recovery code then returns to the bundle after the `chk.s` and the program execution resumes there.

Algorithm 4 Control speculation example.

Without Speculation	Cycle	With Control Speculation	Cycle
		`     ld8.s r3=[r2] ;;`	-X-1
		`     add r4=8,r3 ;;`	-1
`(p1) br.cond label`	0	`(p1) br.cond label`	0
`     ld8 r3=[r2] ;;`	0	`     chk.s r3,recover`	0
`     add r4=8,r3 ;;`	X	`back:`	
`     shladd r6=r5,2,r4`	X+1	`     shladd r6=r5,2,r4`	0
		`recover:`	
		`     ld8 r3=[r2] ;;`	
		`     add r4=8,r3`	
		`     br back`	

It is also possible to speculate uses together with the load; these instructions then must also be replicated in the recovery code. Alg. 4 shows an example of a load with latency `X` that is speculated with its use `add r4=8,r3`: The latency of the load and the add can be completely hidden if they are both hoisted across the branch (under the assumption that their execution can be overlapped with other code before the branch—not shown in the example).

Note that, strictly speaking, the purpose of the recovery code is here not to recover from a failed control speculation, as the name suggests: The speculation fails in the example if the branch to `label` is taken since then the computation of `r4` was unnecessary. The check and the recovery code only ensure proper exception handling in the case of successful speculation. This is different for data speculation introduced in the next section.

NaT bits *propagate*: All instructions set the NaT bits of all destination registers if at least one source register is NaT. Otherwise these NaT bits are always cleared by default (except for speculative loads, of course, which are the only NaT-*producing* instructions). The same rules apply to the NaTVals of the floating-point registers. NaTs are even propagated by transfer instructions that move data between the general purpose and floating-point registers.

The purpose of this propagation is as follows: If several loads are speculated together with a sequence of dependent instructions, it is sufficient to check the result(s) computed by this sequence to detect a deferred exception of any of the loads[2]. The recovery code then repeats the whole speculative computation (with non-speculative loads). This can become a nontrivial problem if register values used in the computation are no longer available at the point of recovery. To ensure recoverability, it can be necessary to enforce the availability or reconstructability of these values throughout speculative computations.

If a non-speculative load or store receives a NaT as address operand, the program terminates with a NaT consumption fault. The same happens if it is attempted to store a NaT—as long as the store has not the completer `spill`: Then the NaT bit is saved in a special register and can be restored by a load with completer `fill`. These instructions are used during context switches.

2.1.5.2 Data Speculation

Data speculation denotes the hoisting of loads above potentially memory dependent (*aliased*, *ambiguous*) stores. In some cases, the compiler might be unable to prove statically that the accessed memory locations do not overlap. Then it is possible to speculate that no aliasing with the store occurs by executing the load as an *advanced load* `ld.a` before the store. Such an advanced load executes like a normal load but allocates in addition an entry with the register number and the memory address in a hardware structure called *Advanced Load Address Table* (*ALAT*). The presence of this entry in the ALAT signals that the corresponding memory location has been read by an advanced load and not been written afterwards. Consequently, any succeeding store invalidates (i.e., removes) all entries representing a memory location that overlaps with the one modified by the store.

After the store, a check load `ld.c` must be scheduled with the same operands as the advanced load in order to verify that no aliasing has occurred (otherwise the advanced load would have read incorrect data). For this purpose, it searches the ALAT for an entry with the same register number and type. If such an entry (still) exists, execution continues normally, otherwise the speculation has failed and the `ld.c` reissues the load.

Alg. 5 shows an example where the use of an advanced load removes the load latency `X` from the critical path. Remarkably, the check load has a zero-cycle latency to consuming instructions and hence can be scheduled in the same instruction group *before* them (here the `add`). Only if it misses the ALAT it incurs a penalty, which may include a pipeline flush (numbers for the Itanium 2 are given in Sec. 2.2.4).

[2]For example, it would also possible—but not advantageous—to check `r4` in Alg. 4.

Algorithm 5 Breaking a memory dependence with an advanced load.

W/o Speculation	Cycle	With Data Speculation	Cycle
		`ld8.a r4=[r3] ;;`	-X or earlier
`st8 [r1]=r2`	0	`st8 [r1]=r2`	0
`ld8 r4=[r3] ;;`	0	`ld8.c r4=[r3]`	0
`add r4=8,r4 ;;`	X	`add r4=8,r4 ;;`	0
`shladd r6=r5,2,r4`	X+1	`shladd r6=r5,2,r4`	1

Algorithm 6 Data speculation with recovery code. Speculating the use `add r4=8,r4` with the load saves another cycle compared to Alg. 5.

W/o Speculation	Cycle	With Data Speculation	Cycle
		`ld8.a r4=[r3] ;;`	-X-1
		`add r4=8,r4 ;;`	-1
`st8 [r1]=r2`	0	`st8 [r1]=r2`	0
`ld8 r4=[r3] ;;`	0	`chk.a r4,recover`	0
`add r4=8,r4 ;;`	X	`back:`	
`shladd r6=r5,2,r4`	X+1	`shladd r6=r5,2,r4`	0
		`recover:`	
		`ld8 r4=[r3] ;;`	
		`add r4=8,r4`	
		`br back`	

If uses are speculated together with the load, the *advanced load check instruction* `chk.a` must be used in place of the check load. Instead of simply reissuing the load, the `chk.a` branches on an ALAT miss to recovery code that reexecutes the load and its uses (see Alg. 6). The whole procedure is very similar to control speculation. Control and data speculation can even be combined using a *speculative advanced load* `ld.sa` that performs all the operations of both an `ld.s` and an `ld.a`. An ALAT entry will not be allocated if this load defers an exception, so a `chk.a` is sufficient to check for both conditions.

A hardware implementation may realize the ALAT functionality incompletely for complexity or performance reasons. These limitations are always designed in such a way that the ALAT may only err on the right side, i.e., they may only cause unnecessary recoveries, but must not suppress a necessary recovery (they are detailed for the Itanium 2 in Sec. 2.2.4). Thus they can never harm the correctness, but only the performance. However, since the ALAT functionality is integrated with the critical L1 cache access path, a simplified ALAT may be crucial to enabling shorter cycle times and thus higher core frequencies.

Frequent ALAT misses—whether due to ALAT limitations or aliasing—can cause penalties that outweigh the benefit of data speculation. Thus the proper use of this feature in a static compiler relies heavily on static analyses: Firstly, the compiler should employ extensive alias analysis to hoist loads above stores even without data speculation [GLS01]. Secondly, a kind of *probabilistic alias analysis* is needed that provides estimates of the aliasing probability for may-aliases (the factor p from Sec. 2.1.5) [JCO98, HCLJ01]. However, it may be considered as a challenge to obtain such estimates reliably through static analysis—a critic puts it drastically [Hop00]: " If the compiler gets the probabilities wrong, the results will be terrible."

2.2 The Itanium 2 Microarchitecture

The Itanium 2 is intended for high-end servers and workstations running enterprise and engineering applications. Its highly complex microarchitecture—designed by hundreds of engineers—is optimized for large-scale, compute and data intensive workloads. The following exposition is the essence of several released documents [BMS02, SR03, FO02, NH02, LDMM02, RG02, Int04]. It covers the most prominent features and achievements of the design and details at the same time all aspects related to instruction scheduling.

2.2.1 Architectural Overview

The newest implementation of the Itanium 2 processor runs at up to 1.6 GHz in a 130nm process with six copper interconnect layers. The power dissipation of its 410 million transistors is about 110W on a typical server workload [SR03].

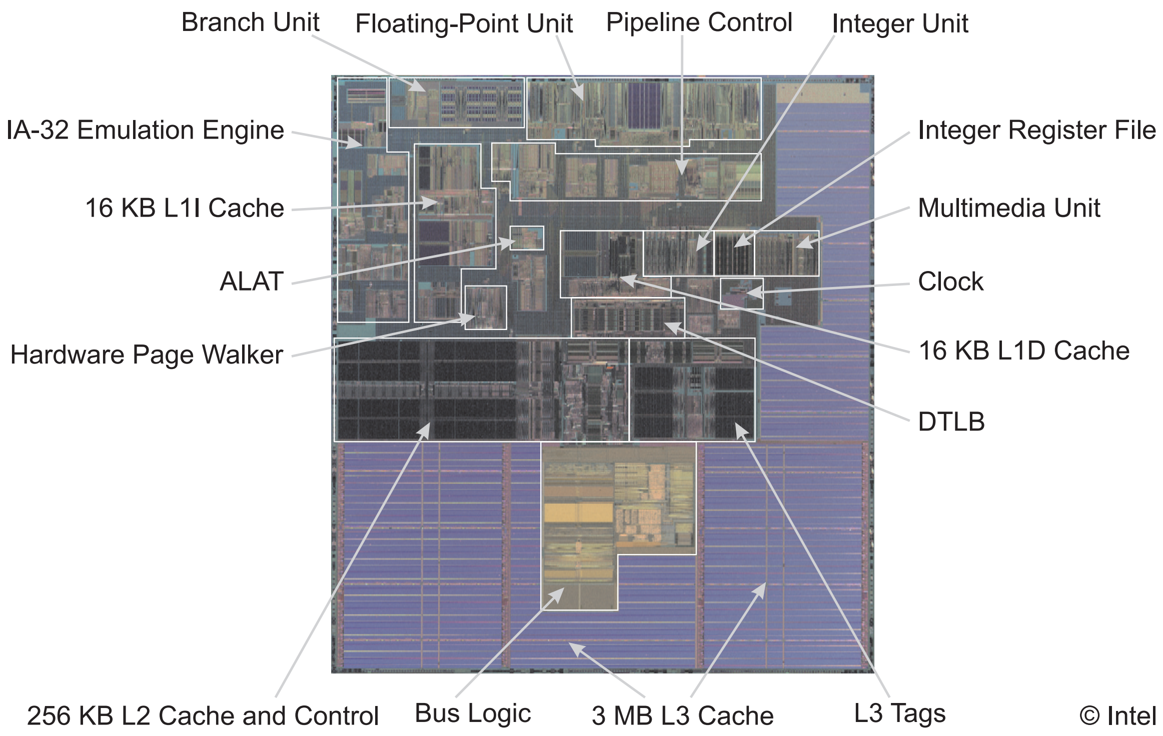


Figure 2.5: Itanium 2 die photo (180 nm version).

Like the first generation, the processor is a six-issue, in-order design: It can fetch, issue, execute, and retire two bundles with six instructions in parallel. The execution of instructions is always *started* in-order (i.e., in the order defined by the instruction groups), but can *complete* out-of-order [HP03]: for instance, the instruction stream can continue to execute while, in the background, a load is waiting for data from the cache. However, if an instruction reads the destination register of this load, then the execution core *stalls* until the cache has delivered the operand. The processor uses a *scoreboard* to track and resolve register dependences [HP03]. Due to out-of-order completion, the number of *in-flight* (actively executing) instructions can be much larger than the issue width of six.

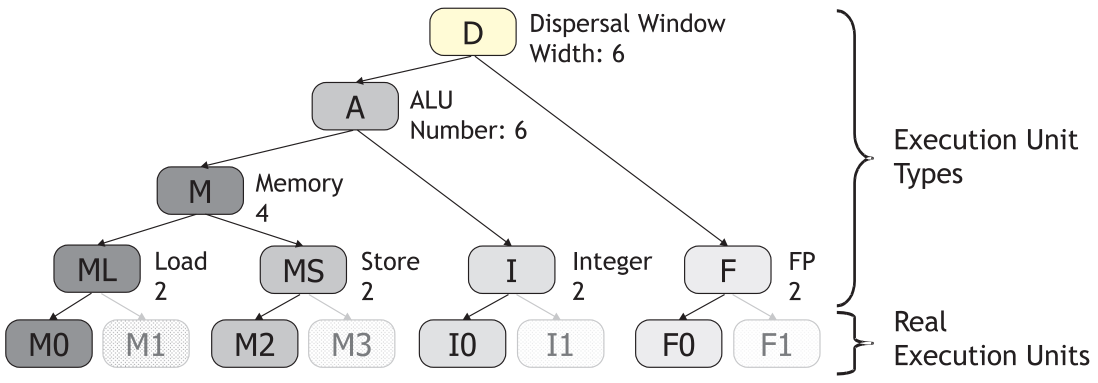


Figure 2.6: Hierarchy of Itanium 2 execution units.

The Itanium 2 features 11 execution units (composed of several subunits): 4 M-units (denoted M0, M1, M2, M3), 2 I-units (I0, I1), 2 F-units (F0, F1), and 3 B-units (B0, B1, B3). Six A-type instructions can be executed per cycle on the M- and I-units; however, the units of these two types are not completely symmetrical otherwise:

- Integer loads can only be executed on M0 and M1 and stores only on M2 and M3. Accordingly, the latter units can be grouped in a set $\mathsf{ML} := \{\text{M0}, \text{M1}\}$, the former in $\mathsf{MS} := \{\text{M2}, \text{M3}\}$.

- Floating-point loads execute on $\mathsf{M} := \{\text{M0}, \text{M1}, \text{M2}, \text{M3}\}$, i.e., on all four memory units (if they are not advanced or check loads, otherwise only ML).

- Control speculation checks can be delivered either to MS or to $\mathsf{I} := \{\text{I0}, \text{I1}\}$. Data speculation checks execute like integer loads.

- Instructions from the groups `FRFR` and `ALLOC` can only execute on M2; `TOFR` only on MS.

- Instructions from `ISHF`, `FRBR`, `TOBR`, `FRAR`, `TOAR`, `FRPR`, `TOPR`, and `MMMUL` can only execute on I0.

There are numerous, further restrictions for special M-type and I-type instructions that are detailed in [Int04]. The floating-point and branch units are almost symmetrical.

The sets ML, MS, M, I, $\mathsf{A} := \{\text{M0}, \text{M1}, \text{M2}, \text{M3}, \text{I0}, \text{I1}\}$, and $\mathsf{F} := \{\text{F0}, \text{F1}\}$ group real execution units that can process a common set of instructions; they can be regarded as *execution unit types* or *abstract execution units*. We can *associate* to each instruction one execution unit type that refers to a set of exactly those real units where the instruction can be executed. The number of instances of an execution unit type equals the cardinality of the set. It is possible to arrange real and abstract execution units in a hierarchy, as shown in Fig. 2.6, where the former are the leaves. In this hierarchy, each abstract execution unit encloses its successor units, i.e., if an instruction can be executed there, then it can also be executed on all successor execution unit types.

Sometimes, we also write single real execution units as types, like $\mathsf{I0} := \{\mathrm{I0}\}$. We often say that an instruction is of (sub-)type ML, MS, M2, or I0 if it is associated to these unit types—however, it should be noted that in contrast to the instruction types defined by the ISA (as in Sec. 2.1.1.1), this classification depends on the microarchitecture.

As a consequence of the asymmetries, there are cases where not all M-type or I-type instructions can be executed on all execution units of the corresponding unit type (different from Fig. 2.1). The instruction issue logic of the processor automatically ensures that instructions are always delivered (“dispersed”) to proper execution units. However, the compiler should anticipate this dispersal process to avoid performance penalties due to resource oversubscription (explained later in Sec. 2.2.3.2).

2.2.2 Cache Design

The Itanium 2 microarchitecture features a three-level, on-chip cache hierarchy. Fig. 2.7 depicts the caches with their respective sizes, associativities, read/write policies, read latencies in cycles (minimum values for the L2 and L3 cache), and peak read bandwidths at 1.5 GHz. While the L1 cache enforces a write-through policy (WT, all writes go directly through the cache to the next-lower cache level), the L2 and L3 caches use write-back (WB, lines are written to the next-lower cache level only on replacement) together with write-allocate (WA, a cache line is allocated also on a write miss). All floating-point memory accesses bypass the L1 cache and are served directly by the L2 cache; they take one additional cycle for format conversion.

The four-ported **L1 data cache** on the top is extremely fast with a single-cycle read latency, which helps avoid load-use stalls of the in-order execution pipeline [BMS02]. The low access time has been achieved through a small cache size (16 KB), aggressive circuit techniques and a *prevalidated tag cache design*.

The latter technique speeds up the translation from virtual to physical addresses, which is necessary as the cache is physically-addressed. The key idea is that the tag array of the cache does not contain the upper bits of a physical address (as usual), but instead a 32-bit pointer to a TLB (translation lookaside buffer, [HP03]) entry that contains this address. This pointer is organized as a “one-hot” vector, i.e., with exactly one bit equal to one, to enable a fast comparison with other pointers. If the i-th bit is set, it is meant to point to the i-th TLB entry. Each L1 cache access then initiates three parallel accesses to different structures:

- The upper-order virtual address bits are used to access the 32-entry L1D TLB, delivering an one-hot vector that points to the entry containing the translated physical address (if existing).
- The lower-order virtual address bits (which do not have to be translated since the way size (4 KB) is always less than or equal to the page size (4 KB-4 GB)) are used to access all four

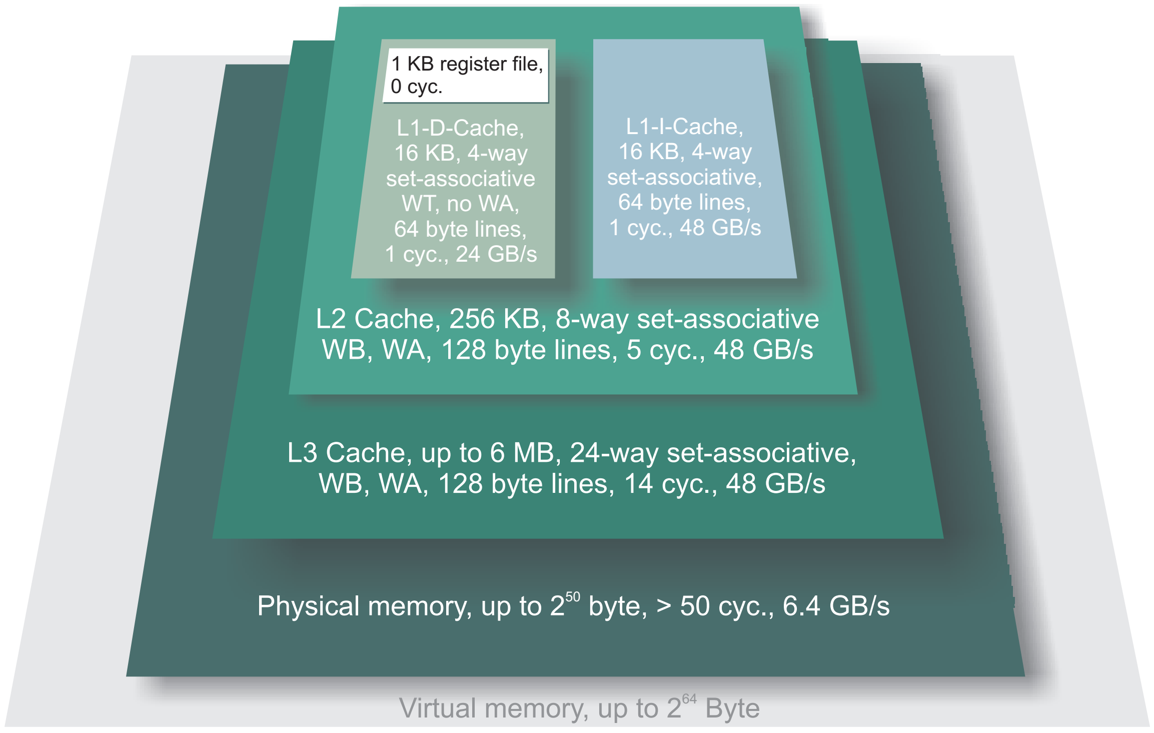

1 KB register file, 0 cyc.
L1-D-Cache, 16 KB, 4-way set-associative WT, no WA, 64 byte lines, 1 cyc., 24 GB/s
L1-I-Cache, 16 KB, 4-way set-associative, 64 byte lines, 1 cyc., 48 GB/s
L2 Cache, 256 KB, 8-way set-associative WB, WA, 128 byte lines, 5 cyc., 48 GB/s
L3 Cache, up to 6 MB, 24-way set-associative, WB, WA, 128 byte lines, 14 cyc., 48 GB/s
Physical memory, up to 2^50 byte, > 50 cyc., 6.4 GB/s
Virtual memory, up to 2^64 Byte

Figure 2.7: The Itanium 2 cache hierarchy.

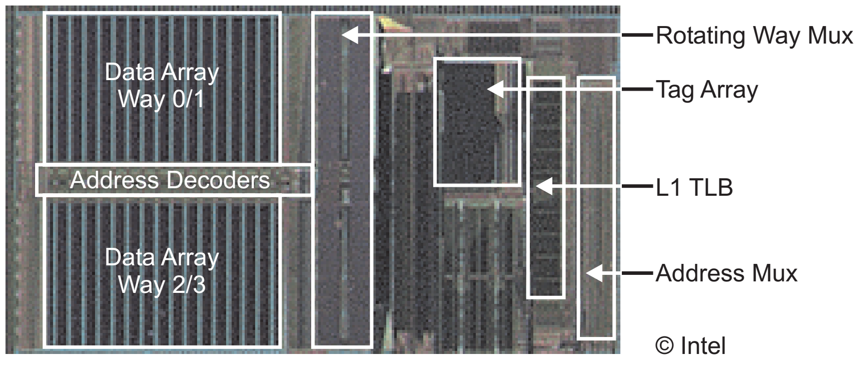


Figure 2.8: L1D cache die photo.

ways of the tag array and the data array in parallel, yielding four 32-bit one-hot vectors and four cache lines, respectively.

The four one-hot vectors then can be compared with the one from the TLB very fast (relative to the comparison of full 64-bit words). The matching one, if existing, determines the way and selects one of the four cache lines for output. The same design is also used in the 16 KB L1 instruction cache.

The multi-banked **unified L2 cache** is a complex, non-blocking out-of-order design. All memory operations that access the L2 cache (L1 misses and all stores) allocate into a 32-entry queuing structure. In each cycle, up to four independent and non-bank-conflicted requests are selected from this queue and issued to the L2 array. The issue logic enforces all architectural memory ordering requirements (semaphore instructions etc.), thus the L2 queue can be regarded as the "central clearing house for all address transactions" in the memory hierarchy [RG02].

The dynamic nature of the L2 cache design makes a precise specification of the access latency impossible. Each read that passes through the L2 queue takes at least 9 cycles. However, there is a feature that allows a request to bypass the queue and issue directly to the L2 data array (provided, inter alia, that there are no dependences on older operations in the queue), enabling a 5- or 7-cycle read latency. These bypasses and further details of the L2 cache design are described in [Int04].

Loads can additionally be delayed if they cause TLB misses: Loads that miss in the L1 DTLB also miss in the L1D cache; if they hit in the L2 and in the 128-entry L2 DTLB, they incur a 4-cycle-penalty in addition to the L2 cache latency. An L2 DTLB miss initiates a hardware page walker (HPW) to perform page look-ups, which costs at least 25 cycles.

Other considerable performance penalties can arise from interferences of ambiguous memory accesses in the cache system. This can happen if several loads and stores that access overlapping memory areas are issued in the same cycle (or in consecutive cycles). In these cases, the penalties with respect to the L1D cache are as follows:

- There will be no conflicts between two loads, or between a load preceding a store in an issue group, if the load(s) hit(s) in the L1D.

- If a store precedes a memory dependent load, the store data must be forwarded to the load. This costs 17, 3-5, 3, and 1-3 cycles if the store is executed 0, 1, 2, and 3 cycles before the load, respectively. In the first two cases, only the lower 12 bits are used for the address comparison. The 17 cycle delay occurs since both requests are passed to the L2 in this case and conflict with each other there. To avoid these penalties completely, the store and load must be separated by at least four cycles.

- Two stores can conflict under circumstances described in [Int04] since the L1D is only pseudo-dual ported for write accesses. Then the younger store will wait in a store buffer; the L1D will stall if this buffer is full.

It is important that these penalties are allowed for during scheduling [CL03]. The L2 conflict conditions are detailed in [Int04].

The **large on-chip L3 cache** is optimized for density. It consumes more than half of the processor area and is tiled into 140 subarrays to fit the irregular shape of the core. It is a pipelined, non-blocking design that has its own queue to support up to eight outstanding request. The minimum read latency is 14 cycles [SR03].

The extensive cache system reflects the necessity to minimize the load latencies on this in-order processor—in total, up to 54 accesses can be active throughout the memory hierarchy without stalling the execution pipeline. The design team focused their resources on the cache hierarchy instead of the pipeline, which has a relatively simple design [MS03].

2.2.3 Pipeline Design

The simplicity imperative of the EPIC philosophy is reflected by the Itanium 2's straightforward and—in relation to the frequency target—short pipeline with eight stages (see Fig. 2.9; the last depicted pipeline stage FP4 is only for floating-point instructions). It is composed of two parts: The two-stage front end, where instruction fetch and branch prediction occur, and the six-stage back end, where instructions are decoded and executed ("execution pipeline"). Both parts are decoupled via an eight-bundle instruction buffer that can be filled by the front end even if the back end is stalled.

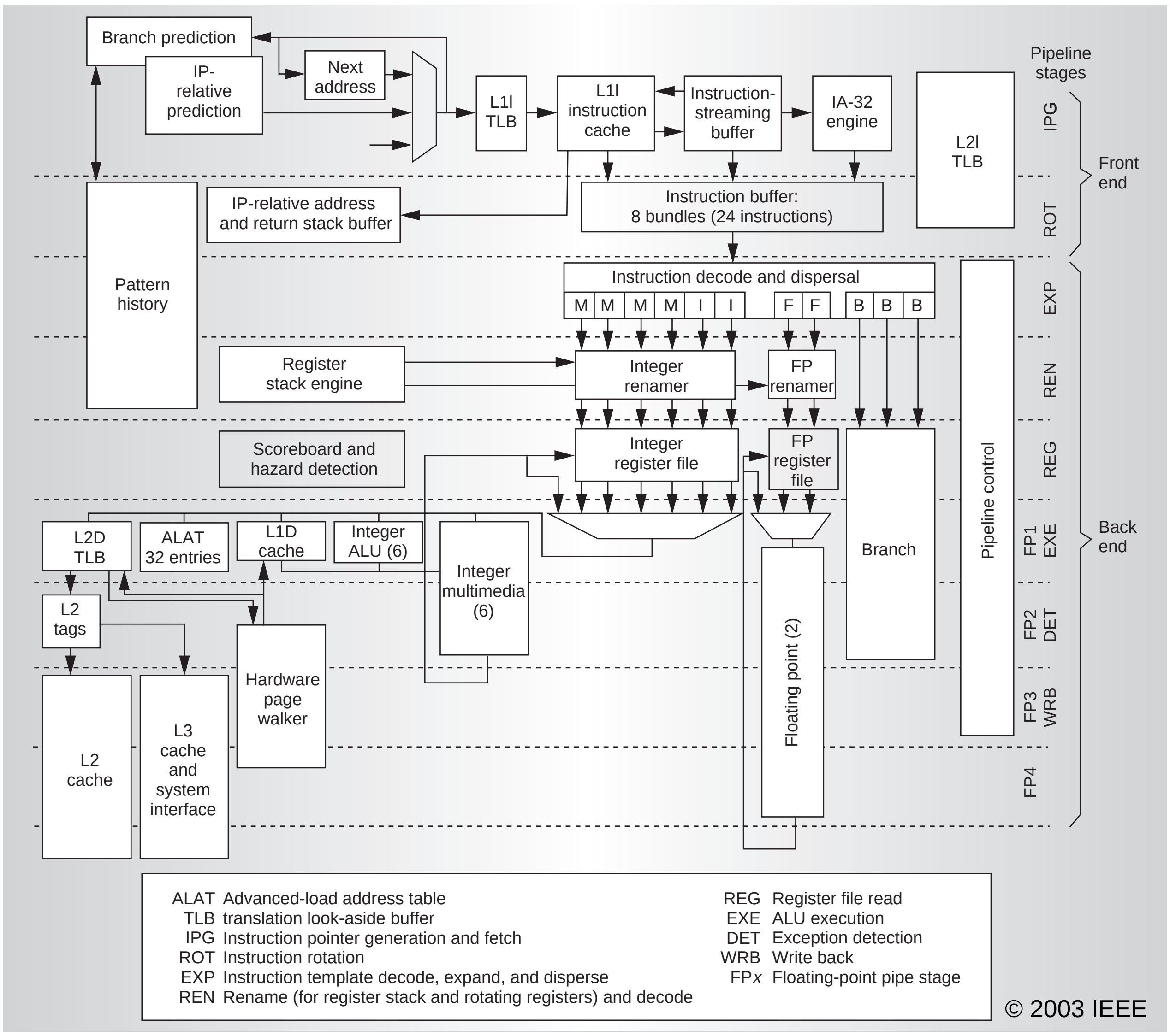
2.2.3.1 Front End

The first pipeline stage IPG generates the *instruction pointer* (IP) and accesses the L1 instruction (L1I) cache. It chooses as the IP either the next linear IP or a *resteer pointer* delivered by the branch prediction logic (which can result from a predicted branch, or from the correction of a mispredicted branch).

The design features two levels of branch prediction: The first level is tightly coupled to the L1I cache. A set of two bundles (32 bytes) is read from this cache per cycle. Each such set is accompanied by a branch target address (shared by all six instruction) and branch prediction information (for each branch instruction in the two bundles). This information is processed according to the Yeh-Patt algorithm [YP91]. If the branch to which the branch target address belongs is predicted taken, then this address is available already in the next cycle (zero-cycle resteer). Taken IP-relative branches that are correctly predicted this way incur no *pipeline bubbles* (i.e., penalty cycles where the IPG stage does not produce a result). If only the target address is predicted incorrectly, the penalty is one cycle. An incorrectly predicted branch *direction* (taken/not taken), however, incurs a six-cycle penalty (see Sec. 2.2.3.3).

The small first-level branch prediction storage (1,000 entries) is backed up by an L2 branch victim cache (24,000 entries) that stores the branch prediction information that is evicted from the first-level structure. There are also two levels of TLBs solely serving instruction access.

The next stage ROT rotates the bundles in order to align them for use in the next stage. This is necessary because the L1I cache is always accessed on even bundle (i.e., 32-byte aligned) address boundaries. If a branch targets an odd bundle address, then the L1I cache is accessed with the next lower even bundle address, with the consequence that the first of the two fetched



ALAT	Advanced-load address table	REG	Register file read
TLB	translation look-aside buffer	EXE	ALU execution
IPG	Instruction pointer generation and fetch	DET	Exception detection
ROT	Instruction rotation	WRB	Write back
EXP	Instruction template decode, expand, and disperse	FP*x*	Floating-point pipe stage
REN	Rename (for register stack and rotating registers) and decode		

Figure 2.9: Itanium 2 processor pipeline (courtesy of IEEE [MS03]).

bundles is useless and discarded in the ROT stage. Hence branch targets should be aligned on 32-byte boundaries to ensure that the front end can deliver two bundles per cycle.

After rotation, the (maximally) two bundles are stored in the instruction buffer if the latter is nonempty or if the back end is stalled; otherwise they are directly forwarded to the next stage EXP.

2.2.3.2 Instruction Dispersal

The EXP (expand) stage is the place where the instructions are extracted from the bundles. For this, the stage logic decodes the bundle templates (and partly the instructions, to allow for the asymmetries) and disperses the instructions to the 11 *issue ports* (illustrated in Fig. 2.10). These ports allocate them to the corresponding 11 execution units two stages later.

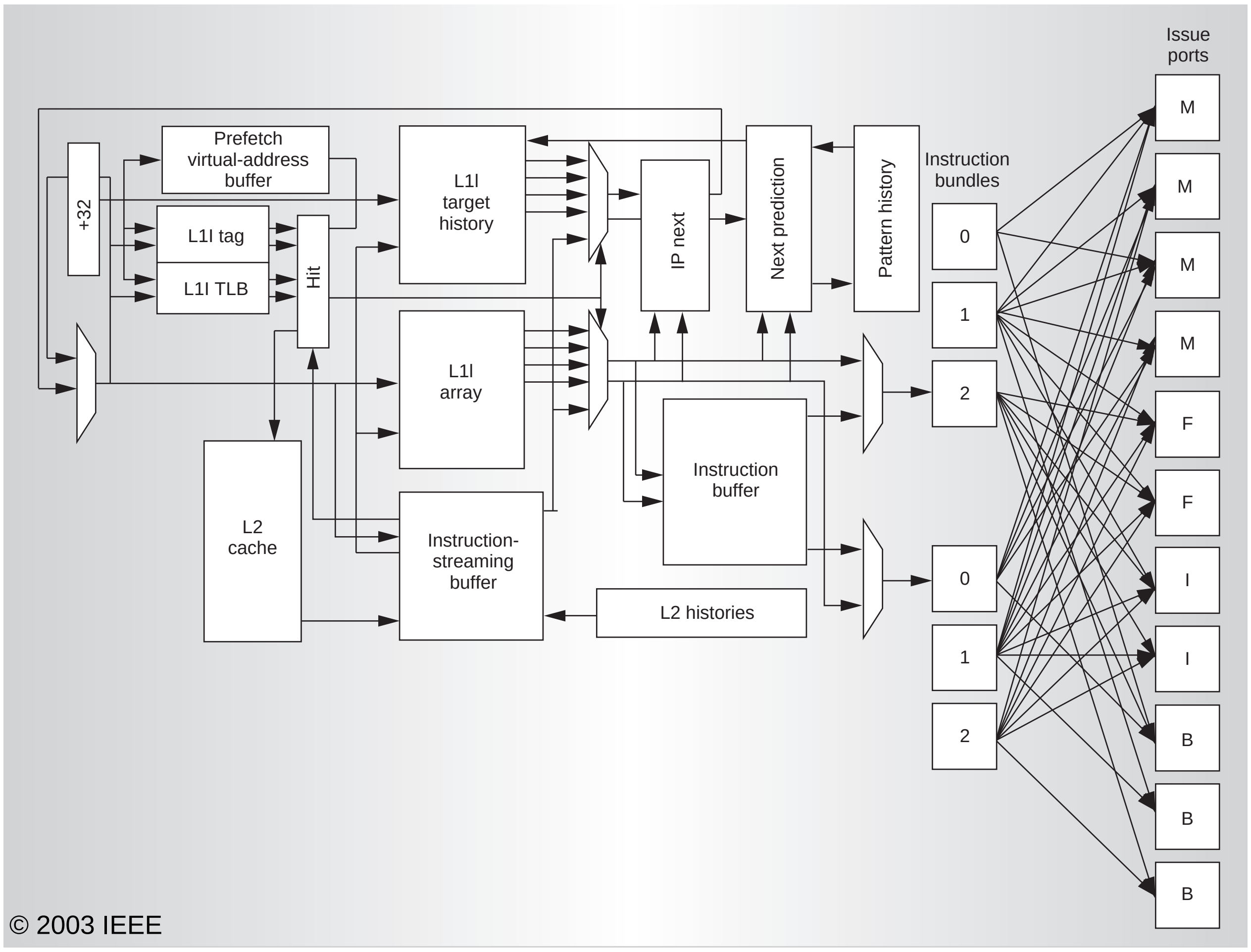


Figure 2.10: Itanium 2 processor front end and dispersal logic design (courtesy of IEEE [MS03]).

The dispersal logic views at most two bundles at a time (the so-called *dispersal window*). During one cycle in the stage EXP, the instructions in the window are always issued in their order (as defined in Sec. 2.1.1.2), schematically one after the other, until

- there is no free issue port available for the next instruction, or
- the end of the dispersal window has been reached, or
- a stop is encountered.

Then all predecessors of the last processed instruction in the dispersal window have been issued, but none of its successors. The latter instructions remain in the window and wait for dispersal in the next cycle. Hence, during each (non-stalled) cycle, between one and six instructions are dispersed in EXP. Accordingly, the processing of either 0, 1 or 2 bundles is completed and the same number of new bundles are brought from the instruction buffer into the dispersal window in order to be considered for issue in the next cycle. Predication does not affect the dispersal process; similarly, nops are issued as if they were normal instructions.

The groups of instructions issued during a cycle in EXP, called *issue groups* or *dynamic instruction groups*, remain constant throughout the rest of the in-order execution pipeline; there is no further splitting or rearranging. If in one cycle the dispersal does not end at a stop, but due to one of the other conditions, then the statically encoded instruction group is not issued as a whole. This is called *split issue*; the first of the above three cases is also called *resource split*. Thus dynamic instruction groups can be smaller than their static counterparts (but never larger). This dynamic adaptation to the processor's execution resources allows—in contrast to VLIWs—binary compatibility between different implementations. However, frequent occurrences of split issue may degrade performance. Therefore, code generators usually produce instruction groups that can be issued in one cycle on the given target processor, so that static and dynamic instruction groups are there equivalent.

To achieve this, the compiler must anticipate the dispersal process; in particular, it must follow the *dispersal rules* to avoid an early resource split. These complicated rules determine to which execution units the instructions are dispersed (depending on the order of their issuing during a cycle in EXP). The most general dispersal rule states that *in most cases*, instructions are assigned one after the other to the lowest numbered matching issue port not already in use. For example, the first I-slot instruction is issued to I0 and the second to I1 etc.

Figure 2.11 shows the dispersal of a four-bundle sequence in three cycles as an example. The bundles and instructions, respectively, are depicted in increasing order from left to right; we distinguish between the older, left (*first*) and the newer, right (*second*) bundle in the dispersal window.

In the first cycle, a resource split occurs after the dispersal of four instructions because there is no free I-unit for the third I-type instruction. The dispersal continues at this instruction in the next cycle after a *single bundle rotation*, i.e., after one new bundle has been inserted from the right side (the greyed out instruction is not considered as it has already been issued).

This time, the dispersal aborts at the stop in the second bundle. Again, a single bundle rotation occurs, and in the third cycle, the remaining fragment from the first bundle and the second bundle can both issue (*dual issue*). Remarkably, the single F-type instruction issues to F1 in this case:

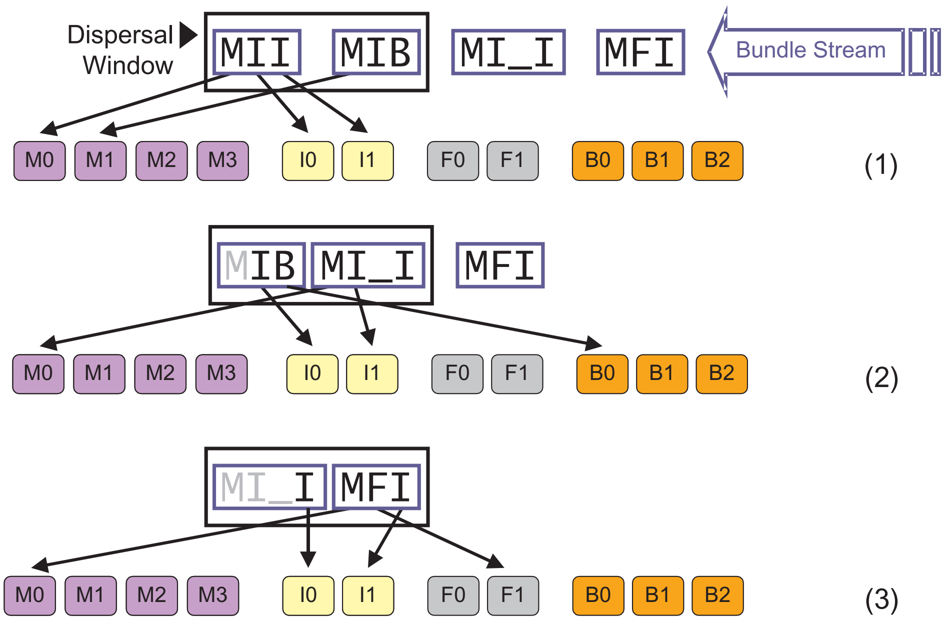


Figure 2.11: Instruction dispersal example.

the F-type instruction from the first and the second bundle always map to F0 and F1, respectively. This is the first exception from the general dispersal rule stated above.

Two further, important exceptions concern A-type and M-type instructions: An A-type instruction in an I-slot can also be mapped to an M-port if the two I-ports are already reserved. This ”trick” is intended to increase the flexibility of the bundling scheme. For example, in Fig. 2.11-(1), if the third I-slot instruction was of type A, then it would issue to M2, and together with the next instruction a dual issue would take place.

The dispersal rules for M-type instructions are the most complicated. The general rule would assume that M-slot instructions are allocated to the ports M0, M1, M2, and M3 in this order; however, this is not flexible enough since it must be possible to issue two loads and two stores in any possible order, instructions which can only be executed on ML $= \{M0, M1\}$ and MS $= \{M2, M3\}$, respectively. To enable this, the issue logic can reorder M-type instructions *between* the subtypes ML and MS. For instance, if an integer store (MS) precedes an integer load (ML), then the store will be mapped to M2 and the load to M0. However, it is not possible to reorder instructions *inside* the same subtype ML or MS, i.e., if an integer store precedes a `getf` (both inside subtype MS), then the store will occupy M2 and a split issue will occur since the `getf` can only issue on M2.

The process becomes intransparent when A-type instructions consume M-ports *before* more restricted subtypes like integer loads and stores; then the dispersal follows irregular, undocumented rules [Int04]. Hence it is strongly recommended that more restrictive subtypes get scheduled first in the instruction group (as long as the intra-group dependences allow this).

There are several further processor-specific special cases where bundle pairs will not experience dual issue: For instance, the issuing splits always after a bundle with a branch instruction: more precisely, it always splits after the bundle templates MBB or BBB and after MIB/MFB/MMB if the B-slot contains a branch (and not a nop or a `brp` instruction). Moreover, dispersal always stops after a bundle with a SIMD FP instruction (as long as the second bundle is not of type MLX; an MLX bundle uses ports equivalently to an MFI bundle otherwise).

Despite the complexity of these rules, the dispersal takes place in the 2/3 ns of one pipeline stage.

2.2.3.3 Execution Core

The REN stage translates the stacked registers into physical registers according to the scheme described in Sec. 2.1.1.3; there is no dynamic register renaming as in typical out-of-order processors [HP03]. Instruction decoding is also performed in this stage.

In the succeeding REG stage, the spills and fills of the register stack engine are injected in the pipeline and the register files are read. The large 128×65 bit integer register file[3] delivers a maximum of 12 integer operands per cycle through its 12 read ports. However, as the write back to the register file occurs three pipeline stages later, the results from the last four cycles are not included in the register file although they have to be considered. Thus there are four stages of bypass multiplexing (utilizing 280 bypass comparators) to select the actual operands among 34 possible results from the register file, the L1D cache, and the next three pipeline stages [FO02].

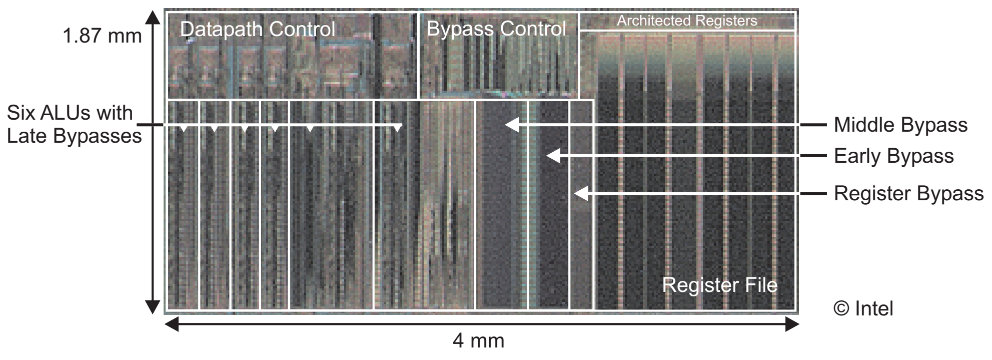


Figure 2.12: Die photo of the six ALUs, the register file and the bypass network.

In particular, the six ALUs are fully bypassed: All simple ALU instructions require a half cycle for execution and a half cycle to forward the result to another ALU or the L1D cache as input of a consumer instruction, totaling an effective single-cycle latency. Fig. 2.12 shows a die photo of this compact 6-issue integer datapath, closely coupled with the register file.

In some cases, bypassing is not free; thus it is advisable to consider the *total latency* of a producer-consumer instruction pair during scheduling, composed of the execution latency of the producer plus the bypass latency, the time it takes to route the result to the consumer. Table 2.7 lists the total latencies of common instruction classes.

[3] 64 data bits plus one NaT bit.

Consumer / Producer	Qualifying Predicate	Branch Predicate	Simple ALU	Load/ Store Address	MM	Store Date	FP	getf	setf
Simple ALU			1	1	2	1			1
MM			3	3	2	3			3
A-Type Predicate Write	1	0							
F-Type Predicate Write: fcmp	2	1							
F-Type Predicate Write: others	2	2							
getf			5	6	6	5			5
setf						6	6	6	
FP						4	4	4	
Integer Load			N	N+1	N+1	N			

Table 2.7: Total instruction latencies.

Most common A-type and I-type instructions (“Simple ALU”) exhibit a single-cycle latency, but multimedia (MM) instructions need two cycles. The latter include integer SIMD instructions and variable shifts (`MMALU_A`, `MMALU_I`, `MMMUL`, `MMSHF`); it can also be seen from the table that bypassing between MM and non-MM instructions takes an extra cycle.

All floating-point instructions, except for compares, have a latency of four cycles. The transfer between general purpose and floating-point registers is expensive with 5-6 cycles. The integer load-to-use latency is, as discussed in Sec. 2.2.2, highly dynamic with a minimum of N=1. If the loaded value is used as the address operand of another memory access, an additional cycle is needed for bypassing.

To steer the bypass network, a register scoreboard is accessed in parallel with the register file. This structure contains for each register whose actual value has not yet been written to the register file *if* and *where* this value is available in the bypass network. In the next cycle, the execution of the issue group is started in the stage EXE, provided that *all* operands are available; otherwise the whole back end of the pipeline stalls (i.e., a single missing operand of an instruction stalls the entire issue group and all subsequent issue groups).

All functional units are fully pipelined and can accept a new instruction each clock cycle, so that the execution pipeline never stalls due to busy execution units (special system instructions may be an exception). Stalls are typically latency-related.

To avoid unnecessary stalls with respect to predication, a producer-consumer instruction pair should never trigger a stall if either the producer or the consumer is predicated off. This rule is implemented with the exception that the execution pipeline will stall for one cycle if

- the operand of an instruction is not yet available and
- the predicate that turns this instruction off was generated in the previous cycle.

The predicate values, which are computed by compares in the EXE stage, are also used to validate conditional branches in the succeeding DET stage. If a misprediction is detected (wrong target, wrong direction), the whole pipeline is flushed and a resteer with the correct branch target address is delivered back to the first pipeline stage. Pipeline flushes cost at least 6 cycles, but the overall impact on performance is moderate due to a prediction accuracy of 93-95% [NH02].

The DET stage offers the last possibility to flush the pipeline since it is the last speculative stage; in the next stage WRB, the issue group is retired, i.e., all results are committed to architectural state: Stores update the data arrays of caches not until now, and results are finally written back to the register files.

2.2.4 Speculation-Related Penalties

Control speculative loads execute like normal loads if they do not set the NaT bit. Otherwise, they take two cycles in the case of an L1 DTLB hit. When a control speculative load misses both DTLB levels, it depends on the configuration of the processor whether it waits for the hardware page walker to retrieve a translation from the page table, which takes at least 25 cycles, or whether the NaT bit is set immediately, although it is not yet clear that the load really causes an exception *(early deferral*) [Int02b].

The advantage of the second strategy is that it limits the worst-case latency of speculative loads. If this is not done, and if a use is speculated with the load, then an HPW invocation would cause the use to stall the pipeline for 25 or more cycles—a *speculative stall* that is futile if the speculation fails (otherwise it is neutral as it would also occur without speculation).

Such speculative stalls can also be caused by cache misses; they should be taken into account when speculating loads with their uses[4]. Ideally, the use of control speculation should be guided by a cost model—available from static analysis, heuristics, and profiling—that estimates the exception deferral probabilities of individual loads. Apart from stall-related issues, control speculative loads have also the potential to pollute the caches and the TLBs.

Many of these difficulties apply also to data speculative loads. The Itanium 2 processor ALAT has 32 entries and is fully associative with respect to the register numbers, i.e., no two such numbers share an entry. If entries are added and the ALAT is full, valid entries are replaced on a FIFO basis (which may result in unnecessary recoveries).

Each entry stores only the 20 lower bits of the physical address, with the consequence that a store falsely invalidates an ALAT entry if their addresses share these bits (but are different otherwise). If the execution distance between a `ld.c`/`chk.a` and a preceding store is one cycle or less, then even only the lower 12 bits will be effectively used for the address comparisons with this store (since the higher bits have not yet been translated to physical addresses at this point of time).

Failed data speculation is expensive: A check load that misses the ALAT takes at least 8 cycles (pipeline flush plus load execution). The pipeline flush becomes necessary because the detection of ALAT misses occurs late in the pipeline in the DET stage, when the incorrect data of the advanced load has possibly already been used in computations. The cost of a `chk.a` is at least 18 cycles for the (always unpredicted) branch to recovery code, plus the cycles needed to execute this code and to return.

[4]In the model of Section 2.1.5, they increase the variable part C_v of the speculation costs.

2.3 The State of Affairs

Since its introduction in 2001, the Itanium Processor Family has established itself as one of the leading processor architectures with respect to performance. However, the superiority in principle of "Intel's huge bet" [ML02] over conventional RISC architectures remains in parts to be proven.

A recent study [Alp03, MK02] has compared various characteristics of the code produced by Intel's C++ compiler 6.0 for the Itanium 2 with that of a classic RISC, the Alpha 21264 (using the Compaq compilers). Basis was the SPEC CPU2000 benchmark [SPE00]. On this benchmark, the Itanium 2 and the Alpha 21264 achieve SPECint/SPECfp base rates of 683/1396 and 621/776, respectively (both at 1 GHz, with the compilers mentioned above) [SPE00]. In the following, all numbers reported from the study refer to the dynamic traces of instructions executed during the benchmark runs. Profiling was used on the Itanium 2, but not on the Alpha.

The traces show that the total number of executed instructions, including nops, is about 20% greater on the Itanium. Without nops, however, the numbers for both architectures are almost equal. The restrictive bundling scheme is the main cause of extra nops. Together with the larger instruction encodings of IA-64 (due to the larger number of registers, predication, and template bits), the total size of the fetched instructions is here about 60% larger than on the Alpha, which inevitably decreases the instruction cache efficiency. Nevertheless, the 16 KB instruction cache is sufficient for SPECint and SPECfp with their relatively high instruction cache locality: the performance loss due to instruction access stalls is just 3% and 1%, respectively. However, for the larger code working sets of server applications, numbers like 31% were reported for the first generation Itanium [Li01].

An analysis of the instruction mix (without nops) shows that the Itanium needs 40% fewer memory operations and 30% fewer branches than the Alpha, but 10% more ALU operations, shifts, and compares. This indicates that features like the large number of architected registers, the register stack engine (RSE), and predication are successful in reducing the number of "hard", stall-inducing instructions like loads and branches. Especially the RSE is seen as an "effective performance enhancement" [Alp03] as it manages procedure calls with very low overhead: The time spent on RSE activity is only about 2% of all cycles (1.5-3 cycles per call/return pair on average).

The effectiveness of predication is more arguable: An earlier study conducted with the first-generation Itanium confirms that if-conversion via predication can reduce the number of mis-predicted branches by 29%—but it improved performance by only 2%, which lags far behind predictions of more than 30% from earlier research studies [CKGN01]. One reason for this is that these studies assumed fewer and less sophisticated branch execution and prediction resources than what materialized on the first-generation Itanium processor. The penalty for mispredicted branches is there only 7% on the SPECint benchmark, which bounds the benefit from removing them. However, this number is likely to increase if the pipeline is stretched on future imple-mentations in order to achieve higher frequencies—hence it is too early for a final verdict on predication. Furthermore, predication is also used by software pipelining: While the benefit of the latter transformation is negligible for SPECint (1%), it is dramatic for SPECfp (more than 30%).

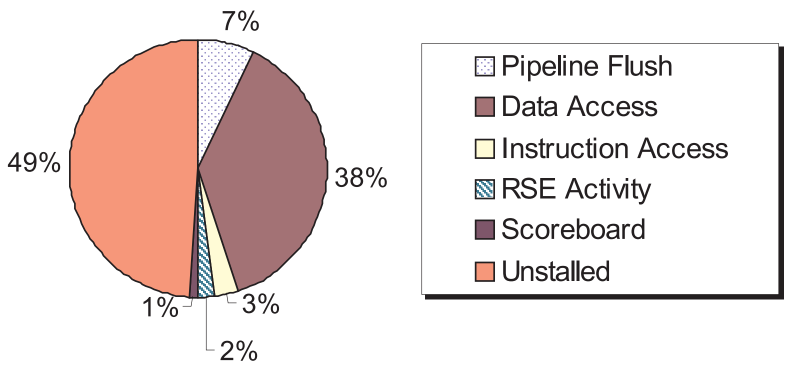


Figure 2.13: Breakdown of the execution time for SPECint 2000 [MK02].

The study on the SPEC benchmarks on the Itanium 2 also includes an analysis of the compiler's ability to extract instruction-level parallelism [Alp03, MK02]. The found static instructions per clock rate (without nops and predicated off instructions) for SPECint is 2.5 on average, far from the maximum rate of 6 IPC the processor was designed for. At runtime, stalls caused by cache/TLB misses and other hazards almost halve this average rate to 1.3 IPC. In other words, the unstalled execution time is about half of the total execution time. This is depicted in Fig. 2.13, where "Data Access" denotes the stalls due to cache-missing loads and "Scoreboard" all stalls due to *other* long-latency instructions. The authors also investigated the region sizes in the trace and measured 19 useful instructions per taken branch, 159 useful instructions per mispredicted branch and 212 useful instructions per call on average.

While the study did not analyze the benefit of speculation, it measured that about 24% of loads use control speculation, but only 4.5% data speculation. The failure rates are very low with less than 0.001% and 1% for control and data speculation, respectively. This shows that the compiler has the cost of speculation well under control, but it leaves open whether this is merely due to conservative compiler heuristics that restrict the application of this feature.

Overall, these numbers paint a rather positive picture: the Intel compiler can transform the architecture's new features into performance that is "highly competitive with the best RISC processors" [Alp03]. But the high percentage of nops and the low static IPC indicate that some central visions of the architecture's inventors have not (yet) come true. Intel's compiler team has identified the following top-five challenges [Li01]:

1. Managing data caches/DTLB for acyclic code
2. Managing instruction cache/ITLB
3. More effective use of control speculation
4. More effective use of data speculation

5. Creative use of predication

This work tackles all of these items, especially the last three, and it also points out another candidate for the list: More effective global instruction scheduling.

Chapter 3

The Global Instruction Scheduling Problem

3.1 Overview of Code Generation

The process of compiling a program can be coarsely structured into an *analysis* and a *synthesis* phase [WM97]. The analysis phase computes the syntactic structure and semantic properties of the input program. The result is an intermediate representation that is (largely) independent of the language and the target machine. The compiler performs several optimizations like constant propagation, elimination of common subexpressions, etc., on this intermediate code.

It is the task of the subsequent synthesis phase to convert the intermediate representation into semantically equivalent target machine code that executes with best possible performance. An important part of this phase is *code generation*, which consists itself of several subphases:

- *Code selection* is the task to find a sequence of target machine instructions for the intermediate code that has the same semantics and minimal execution time. This part is of subordinate importance on the Itanium processor architecture, which features a RISC-like instruction set and a load/store architecture without complex addressing modes. Exceptions may be programs that make extensive use of the SIMD and floating-point instructions.

- *Register allocation* decides where the values of the intermediate representation are kept in the processor registers. Intermediate code uses an unlimited number of symbolic registers that must be mapped to the limited number of architected registers. The goal is to minimize the transfer between registers and memory. Like code selection, register allocation has in general not the highest priority on the Itanium architecture with its very large number of architected registers.
  However, routines that allocate a large register stack frame can put pressure on the register stack engine. This can even lead to RSE-related stalls, which may carry weight with short routines. The register need of software-pipelined loops is also much higher because the values of several overlapped loop iterations must be held in registers at the same time.

- *Resource binding* selects for each instruction the execution unit to which it is issued (on architectures that let the compiler decide this). This subtask is especially complicated if this selection has impact on the instruction's latency or if it restricts the range of registers it can access, which is both not the case on the Itanium 2. Nevertheless, resource binding is still nontrivial here due to several asymmetries (outlined in Sec. 2.2.1).

- *Instruction scheduling* rearranges instructions with the goal to minimize the schedule length. This phase will be elaborated on in the remainder of this chapter.

The order in which these tasks are accomplished is not arbitrary since there are strong interdependences between them: decisions made in one phase may constrain the scope of possible decisions of subsequent phases. Performing the phases in an isolated way cannot take all these interactions into account. This problem is known as the *phase-coupling problem* [Käs00a].

For example, if scheduling is performed first, then the schedule determines the values that are concurrently live at each program point. If their number exceeds the number of available architected registers, further stores and loads have to be inserted that transfer values to and from memory ("*spills*" and "*fills*"), deteriorating the quality of the schedule. Alternatively, if register allocation is done before, it is likely to introduce false dependences between the instructions that confine the scheduling phase. Most compilers perform instruction scheduling first, possibly followed by another pass of scheduling [SS02].

There are various works that tackle the phase-coupling problem directly, either heuristically or using exact approaches; comprehensive surveys are given in [Bas95, Käs00a]. In the remainder of this chapter we will encounter similar interdependences *within* the global scheduling phase; some of them stem from Itanium-specific features like explicit speculation.

The goal of instruction scheduling is, as mentioned earlier, to minimize the overall execution time by reducing the schedule lengths of the basic blocks. These lengths are static measures; the performance impact of their reduction depends on which control flow paths are taken at runtime, which also depends on the input set—the variance resulting from this can be considerable. Static compilers must optimize for the *average case*, ideally by means of *profiling information*. This information provides an empirical summary of past program executions. It typically has to be measured by the developer by running an *instrumented version* of the program before the final compiler run.

The most commonly used types of profiles are *control flow profiles*: Node profiles, edge profiles, and path profiles provide the execution frequencies of basic blocks, control flow edges, and entire control flow paths (acyclic and intraprocedural), respectively. They are listed here in the order of increasing precision and increasing instrumentation overhead. Often, it is possible to identify paths with a dominating execution frequency (*hot paths*).

Profiling is important to guide compiler decisions that involve "trade-offs between improved performance in one part of the program and degraded performance in another part of the program" [SS02]. However, collecting profiles ahead of time is not always practical. If they are missing, the compiler must rely on heuristically generated estimates with low confidence. Besides, profiling cannot mitigate an inherent disadvantage of statically scheduled architectures, namely that it is only possible to optimize for the average case, but not for varying execution characteristics. Dynamic compilation aims to address both problems [SS02].

A further factor that influences performance besides the schedule length is *code size increase*. It must be kept under control by the code generator since it may cause adverse instruction cache and TLB effects. This must occur not only in the code selection phase, but also during global instruction scheduling, as it will be shown in what follows.

3.2 Basic Program Representations

Global instruction scheduling works on bounded, contiguous regions of basic blocks, called *scheduling regions*. These regions constitute the *scheduling scopes* within which the global scheduler can rearrange and parallelize instructions; they should be chosen as large as possible to provide maximal opportunities for the extraction of instruction-level parallelism. Scheduling regions are often—initially also in this work—required to be acyclic (free of loops). Apart from such scheduler-specific restrictions, they are naturally limited by procedure boundaries since code motion between procedures is not considered viable. The notion "scheduling region" is often used with two different meanings:

- It denotes the input the scheduler receives on an invocation during code generation.
- It can also refer to *subregions* formed by the scheduler itself: Some of them partition the given scheduling region further into smaller regions and pass them to a scheduler proper in order to be scheduled. They exhibit a two-level structure: region formation and region scheduling.

The following definitions describe several formal representations of scheduling regions as they are used for scheduling. They are in some details adapted to the Itanium processor architecture:

Definition 3.2.1 (Control Flow Graph) A *control flow graph (CFG)* of a scheduling region is an acyclic digraph $G_C = (V, E_C, V_{entry}, V_{exit})$ with the region's instructions as nodes. The edges represent possible control flow between the instructions and are marked by predicate registers. If an edge $(a, b) \in E_C$ is marked by p then instruction b is executed after a if and only if p has value true. b is then said to be *control dependent* on a. V_{entry} contains those nodes without predecessors (*entry points*) and V_{exit} those nodes without successors (*exit points*). □

In this thesis, assembly instructions form the nodes of the CFG, however, this and the following representations can also be instantiated on a source or intermediate level. Conditional branches are not represented by nodes but only by edges in the graph.

Definition 3.2.2 (Basic Block) A *basic block* in a CFG is a path of maximal length where no inner node has more than one successor or predecessor. □

Those instructions that are nodes on a path of Def. 3.2.2 are said to be *contained* in the basic block embodied by the path. The block is then also called the *source block* of these instructions. The control flow inside a basic block is simple: If during program execution a basic block is reached, then always all of its instructions are executed. To focus on the more dynamic control flow *between* basic blocks, we can regard them as nodes of a new graph, the *basic block graph*:

Definition 3.2.3 (Basic Block Graph) A *basic block graph (BBG)* of a scheduling region is an acyclic digraph $G_B = (\mathcal{B}, E_B, \mathcal{B}_{entry}, \mathcal{B}_{exit})$ with the region's basic blocks as nodes. The edges $E_B \subseteq \mathcal{B} \times \mathcal{B}$ represent possible control flow between the basic blocks and are marked by predicate registers analogously to Def. 3.2.1. $\mathcal{B}_{entry} \subseteq \mathcal{B}$ contains those nodes without predecessors (*entry blocks*) and $\mathcal{B}_{exit} \subseteq \mathcal{B}$ those nodes without successors (*exit blocks*). □

Definition 3.2.4 (Control Flow Paths) Paths in G_C and G_B are called *control flow paths*. They are said to be *complete* if they start from a node in V_{entry} or $\mathcal{B}_{entry}$ and end at a node in V_{exit} or $\mathcal{B}_{exit}$. In the context of a BBG, we denote by $\mathcal{C}$ the set of all complete control flow paths and by $\mathcal{C}(A) \subseteq \mathcal{C}$ the subset of those paths that pass through block A. Complete control flow paths are also referred to as *program paths*. □

In both the CFG and the BBG, nodes with more than one predecessor are called *joins* and nodes with more than one successor are called *splits*. Edges from a split to a join are called *JS edges* [BMM00]. JS edges can complicate scheduling algorithms; it is possible to remove them by adding a new block between the split and the join (called *JS block*). The JS edge is then replaced by two new edges, one from the split to the JS block and one from there to the join.

In the context of basic block graphs, we often use the notions of (direct) successors or predecessors as known from graphs in general: For instance, we say that a block A is a *predecessor* of block B if there exists a nonempty control flow path from A to B, also written $A \prec B$. If this path consists only of one edge, A is called *direct* predecessor. The predecessor relationship $\preceq$ imposes a partial order on $\mathcal{B}$.

Definition 3.2.5 (Dominance, Postdominance) We define for two nodes a and b in an acyclic digraph (typically a CFG or BBG):

- a *dominates* b (a dom b) if every path from an entry node to b passes through a.
- b *postdominates* a (a pdom b) if every path from a to an exit node passes through b.
- a and b are *control equivalent* if a dominates b and b postdominates a (or vice versa).

Each node dominates and postdominates itself. We extend this definition to *node sets* as follows: Let a be a node and S be a subset of nodes, then

- a is dominated by S, denoted by $a \in \mathbb{D}_+^{-1}(S)$, if every path from an entry node to a passes through S.
- b is postdominated by S, denoted by $a \in \mathbb{P}_+^{-1}(S)$, if every path from a to an exit node passes through S. □

Control flow properties can also be described by *control dependence graphs* or *program dependence graphs* [SS02], but these concepts are not used and presented here. Instead, we introduce a further essential graph that describes the data dependences of instructions:

Definition 3.2.6 (Global Data Dependence Graph) Let an acyclic control flow graph

$$G_C = (V, E_C, V_{entry}, V_{exit})$$

be given. The corresponding *global data dependence graph* (DDG) $G_D = (V, E_D)$ is an acyclic digraph that contains an edge from node m to n if

- n is data dependent on m with respect to a storage resource (as defined in Sec. 1.3) and
- there exists a control flow path in G_C from m to n that contains no definition of this storage resource, and in the case of a WAW dependence also no use (this condition is here referred to as the *exclusion criterion*).

The data dependence edges can be partitioned according to the involved dependence and resource types: $E_D = E_D^{RAW} \cup E_D^{WAR} \cup E_D^{WAW}$ (true, anti, and output dependences) and $E_D = E_D^{reg} \cup E_D^{mem}$ (register and memory dependences), respectively. A *(total) latency* w_{mn} in cycles is associated with each data dependence edge; this value is often also interpreted as the *length* of the edge. □

On the Itanium architecture, all intra-group (WAR and memory) dependences have latency zero since the dependent instructions may appear in the same instruction group (though only in an order that complies with the dependence). WAW register dependences, however, cause stalls on the Itanium 2 processor similarly to RAW dependences and are thus assigned the same latencies.

The existence of a DDG edge implies the existence of a (possibly empty) control-flow path between the two instructions' source blocks. Hence acyclic control flow graphs always yield acyclic data dependence graphs. An instruction is called a *DDG predecessor* of another instruction if there is a nonempty path from the former to the latter in the DDG. The order on the instructions defined by the data dependences is transitive, thus a data dependence edge can be regarded as *redundant* if it is already implied by a sequence of other edges. We can assume that a given DDG has no such redundant edges:

Definition 3.2.7 (Minimal DDG) A data dependence graph G_D is called *minimal* if for no edge $(m, n) \in E_D$ there exists a path in G_D from m to n that does not contain the edge (m, n) and has length greater or equal to w_{mn}. □

The DDG is sufficient to describe all feasible orders of instructions contained in a basic block—it renders the CFG, which gives for each basic block a linear instruction sequence (as *one* possible order), dispensable. In other words, the DDG extracts from this linear sequence the information that is relevant for scheduling.

Thus, if we have a function $s : V \longrightarrow \mathcal{B}$ that gives the source block of each instruction, then BBG and DDG together are an (almost) complete description of a global scheduling problem instance. For the example routine from Alg. 2 in Sec. 2.1.2.1, both representations are shown in Fig. A.1 and Fig. A.2 in the appendix.

3.3 Global Scheduling

Local scheduling—introduced in Sec. 1.3—rearranges instructions inside a basic block in order to minimize its length. The basic block boundaries act like barriers: it is not possible to overlap the execution of instructions from neighboring blocks unless instructions are *moved* between basic blocks. Since basic blocks are often small (typically in the range of 5-20 instructions [SS02]), local scheduling is not able to extract enough instruction-level parallelism to keep the parallel execution units of EPIC processors busy.

Therefore *global code motion* is used to move instructions beyond basic block boundaries. We say that an instruction is moved from its source block to a *destination block*, which can be a predecessor or a successor of the source block (other blocks evidently do not make sense as destination blocks). The arrows in Figure 3.1 depict schematically these two possible directions: *upward* and *downward code motion*.

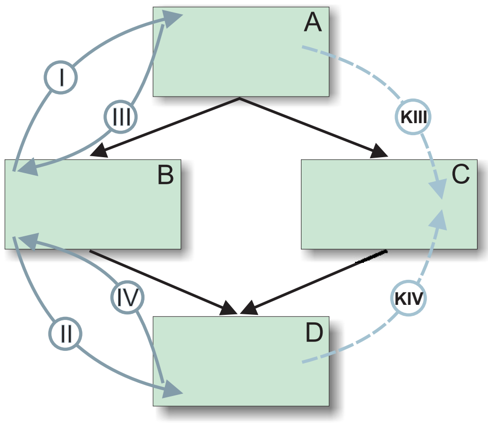


Figure 3.1: Examples of different directions (upward (I+IV), downward (III+II)) and side-effects of code motion (speculative (I+II), non-speculative (III+IV)).

Using this example, we will introduce two basic side-effects of code motion: Firstly, we observe that the upward movement of an instruction from source block B to destination block A (I) is *speculative*: If at runtime the path A-C-D is taken, then the instruction is executed unnecessarily in A (as it would not be executed at all without the movement). It is only *useful* on the path A-B-D. This speculative scheduling has also the consequence that an execution slot is occupied unnecessarily on the path A-C-D. In other words, it increases the demand for execution slots on this path. In contrast, an instruction moved from A to B (III) is always useful since any path through B also traverses A; it is an example of *non-speculative code motion*. Speculative execution is prohibited for some instructions (as described in detail in Sec. 2.1.5 and Sec. 2.1.5.1) so that they may only be moved non-speculatively.

Secondly, an upward movement across a join from D to B (IV) enforces the placement of a duplicate of the instruction in block C (KIV)—otherwise this instruction would not appear on the path A-C-D (as it is the case without the movement). The addition of such *compensation copies* during code motion increases the instruction count and thus also the demand for execution slots.

Generally, when considering code motion along a single BBG edge, upward movement across a join and downward movement across a split enforces compensation copies. Moreover—and independently of required compensation copies—the movement of an instruction is speculative if it occurs upwards across a split or downwards across a join; otherwise it is non-speculative. As it has been shown exemplarily, all these code motion variants increase the *resource demand*, i.e., the number of needed execution slots in the schedule.

We will in Sec. 3.3.2 examine more closely what constitutes speculative code motion and when and where compensation copies have to be scheduled. Before, we will provide a brief survey of existing global scheduling heuristics.

3.3.1 Global Instruction Scheduling Algorithms

Global scheduling is often introduced on the basis of concrete scheduling methods like **trace scheduling** [Fis81], one of the first proposed algorithms. Trace scheduling identifies frequently executed paths in the basic block graph, called *traces*, and schedules these linear sequences of basic blocks as if they were a single basic block, using a local scheduling method like list scheduling [WM97]. During this local scheduling, instructions can be moved beyond the block boundaries of the underlying basic block sequence. When doing this, the compiler must avoid speculative code motion of non-speculative instructions and track where compensation copies have to be inserted (*bookkeeping*).

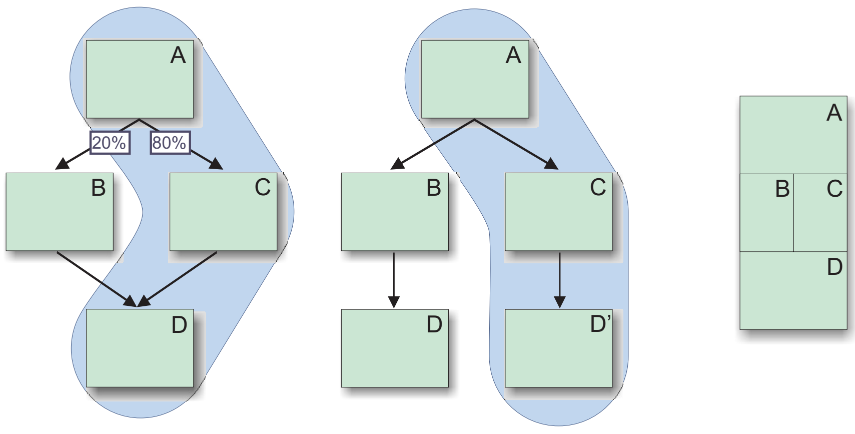


Figure 3.2: Illustration of a trace, a superblock after tail duplication, and a hyperblock.

These additional copies may increase the schedule lengths of other paths—in this way trace scheduling optimizes traces at the expense of off-trace paths, which may be disadvantageous for programs without distinct hot paths. The algorithm first schedules the most frequently executed trace and then selects, in decreasing order of path frequency, traces with unscheduled blocks, until all blocks are covered.

Another drawback of trace scheduling is the complexity of the bookkeeping that is related to joins in the trace (side entrances). Hwu et al. propose **superblock scheduling** to mitigate this complexity [HMC+93]: the selected traces are transformed via tail duplication (i.e., splitting its tail at joins into two different copies) into traces with a single entry and multiple exits. Then upward code motion inside the superblock never enforces compensation copies (the downward variant can possibly be left out without a significant performance loss). Superblock scheduling is more straightforward; however, the tail duplication implicates a code size increase.

A superblock can be extended to include basic blocks from different control flow paths via predication: If it contains an outgoing control flow edge, then the instructions at the branch target can be merged into the superblock if they are guarded by the predicate that controls the branch (*if-conversion*)—the superblock is then called a *hyperblock*. It remains a linear code sequence since if-conversion turns all control dependences into data dependences. The hyperblock exhibits increased parallelism as it contains independent instructions from different paths. This makes the transformation particularly interesting for highly parallel architectures with predication, like EPIC architectures.

The technique that forms and schedules the hyperblocks is known as **hyperblock scheduling** [MLC+92]. Again, tail duplication is performed to keep the hyperblock free of side entrances. As in superblock scheduling, the resulting code size increase may be significant and must be kept under control by the compiler.

All mentioned methods have in common that they tackle global scheduling by reducing it to local scheduling. For this purpose, they select linear scheduling regions on the basis of profiling information. In contrast, the following four methods work directly on scheduling regions with arbitrary acyclic control flow: **Percolation scheduling** [Nic85] applies iteratively four semantics-preserving transformation rules to the control flow graph, three of which perform upward code motion (between adjacent blocks). This method originally assumed unbounded resources and unit latencies (i.e., latencies that are all equal to one) and was extended in further works [SS02].

Gupta and Soffa propose **region scheduling** [GS90] to increase and balance the parallelism in a program. The algorithm works on an extended program dependence graph (EPDG), a *hierarchical* representation of the scheduling region in which the nodes represent individual instructions, predicates or *regions* of control equivalent instructions. Similar to percolation scheduling, the region scheduler repeatedly applies transformations to the EPDG that redistribute code among the regions until no further transformations are possible or the parallelism in each region matches that of the target processor.

The transformations are guided by estimates of the parallelism present in each region (the instruction count divided by the critical path length). They go beyond code motion and also include *loop transformations* (unrolling and invariant code motion) and *region copying and collapsing* (which is basically tail duplication and if-conversion, respectively). Code motion is allowed in both directions, also speculatively or with compensation code. It is not only applicable to the (leaf) nodes of the EPDG that represent instructions, but also to those higher in the hierarchy that represent regions. This means that the scheduler is capable of moving entire subgraphs that represent, for example, an if-statement. In doing so, region scheduling combines scheduling of fine-grain parallelism with transformations of the control structure that expose such parallelism.

Bernstein and Rodeh [BR91] present a comparatively simple global scheduling algorithm that processes the basic blocks inside an acyclic scheduling region in topological order (of the BBG) and schedules them one at a time. The scheduling of each block occurs similarly to list scheduling. In difference to this local method, the list of data ready[1] candidates for scheduling is made up not only of instructions that originate from the block being scheduled, but also of instructions from control equivalent successor blocks and the direct successors of these blocks. In this way, (speculative) upward code motion is incorporated; compensation copies, however, are not supported by the presented implementation. Once all instructions are scheduled that originate from the current block, the scheduler moves to the next block. The approach has similarities with wavefront scheduling, the (patented) technique implemented in Intel's Itanium compiler, which will be outlined now.

Wavefront scheduling [BMM00] uses a high level driver that selects scheduling regions and passes them to the scheduler proper. A scheduling region may contain arbitrary acyclic control flow. After it has been scheduled, it is grouped into one or more new BBG nodes and nested. This is a recursive process that starts at the innermost loops. When regions of completed blocks are nested, the data flow information within them (memory references, live-out and live-in values, outgoing latencies) is summarized in order to allow semantically correct code motion across the resulting nested nodes. In this way, code can even be moved across a loop that is abstracted away through nesting.

A specialty of wavefront scheduling is that it makes extensive use of *path vectors* to represent control-flow related information. Let $\mathcal{C} = \{P_1, \ldots, P_k\}$ be the set of complete paths through the scheduling region, then each subset of $\mathcal{C}$ can be represented by a vector $x \in \{0,1\}^k$ where x_i is equal to one if and only if the subset contains P_i. For instance, $BPV(A) \in \{0,1\}^k$ denotes for each basic block A the subset of paths that flow through it. $Prob(x)$ is the aggregate probability that the control flows along one of the paths embodied by the path vector. Set operations like "$\cap$" and "$\setminus$" can be represented by performing simple boolean operations on the path vectors. The size of path vectors can grow exponentially with the region size—hence the regions are selected in such a way that the number of paths does not exceed a certain threshold.

We now describe how a region is scheduled. The scheduler processes the blocks in the region *top-down* in a certain topological order. This order is defined by the movement of a *wavefront*, a (changing) set of blocks such that each complete path in the region passes through exactly one block in it. The wavefront can be regarded as the *boundary* between scheduled and yet to be scheduled blocks in the region. More precisely,

- the blocks above the wavefront have been scheduled,
- the blocks on the wavefront are being scheduled (simultaneously), and
- the blocks below the wavefront still have to be scheduled.

The initial wavefront consists of all entry blocks. Once the scheduling of a block on the wavefront is finished, the scheduler attempts to advance the wavefront across it[2]. Fig. 3.3 depicts how a wavefront can advance down the region until it passes through all the exit nodes (W1-W6).

[1] An instruction is data ready if it is not data dependent on another, yet unscheduled instruction.

[2] It is noteworthy that the algorithm requires all JS edges to be removed via JS blocks. Only then it is guaranteed that a block-wise advancement of the wavefront is always possible.

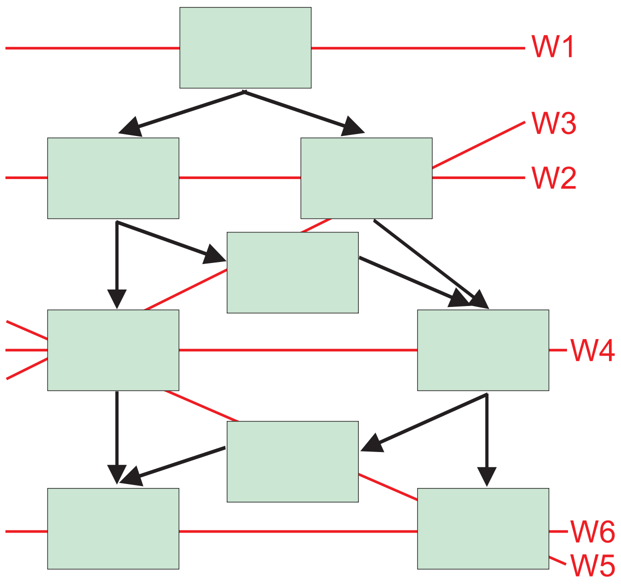


Figure 3.3: Wavefront advancement in six steps.

By scheduling instructions only into blocks on the wavefront, it is guaranteed that the compensation copies they require can be inserted entirely into other blocks on the wavefront. The scheduler tracks the compensation need of each instruction by means of a *compensation path vector*. This vector for an instruction n with source block $s(n)$ is initialized to $CPV(n) := BPV(s(n))$. When scheduling n into a block D, it is changed to $CPV(n) := CPV(n) \setminus BPV(D)$ to update the information where further copies still have to be scheduled. Fig. 3.4 shows an example where $CPV(n)$ is changed from 11100 to 01100 after $n := \mathtt{op}_1$ is scheduled into D.

The generation of compensation copies does not have to occur immediately, instead it can be *deferred* until a better opportunity arrives in a later wavefront (in Fig. 3.4 along the wavefronts W4-W5). Such an opportunity can be a free execution slot in a block for which no other instruction candidate can be found.

However, the compiler must also keep track of the latest feasible blocks where these copies *can* be scheduled: In the example, wavefront W5 is the last possibility. There are other factors that can constrain the downward movement of an instruction, such as the availability of its qualifying predicate. In all these cases, the scheduler ensures that the wavefront is only advanced if all instructions that cannot be deferred further *have been* scheduled (then their compensation vector is $\overrightarrow{0}$). Scheduling multiple copies of an instruction on a path is also avoided.

The actual choice *which* instruction is scheduled next into a block is performed similarly to list scheduling [WM97, Muc97]. For each block on the wavefront, a list of candidate instructions is maintained that contains unscheduled, data ready instructions that originate from a predecessor or a successor of the block. The scheduler selects one of these instructions for scheduling into the block based on a *cost-benefit analysis*. Instructions are preferred that lie on a global critical

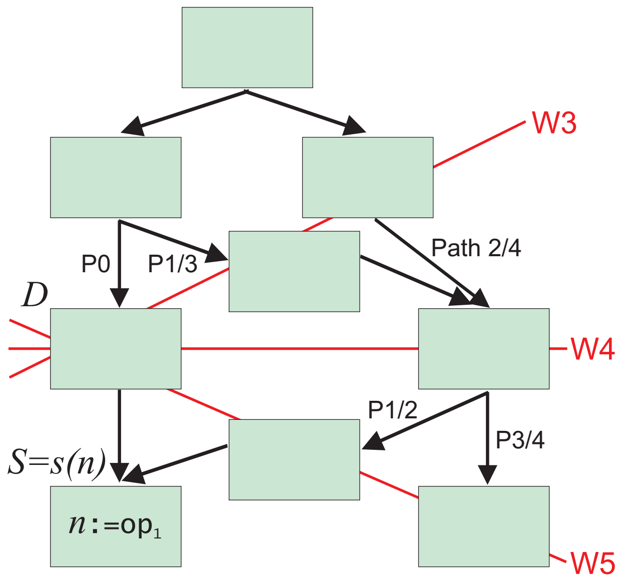


Figure 3.4: Deferred compensation.

DDG path, and—in the case of a speculative candidate—are likely to be *useful* if scheduled speculatively into the block. The *usefulness* is the likelihood that the speculative execution is not futile, namely:

$$\frac{Prob\,(BPV(D) \cap BPV(s(n)))}{Prob\,(BPV(D))}$$

which is the probability that the control flow passes through the instruction's source block $s(n)$ if it flows through D. The cost term of the selection function takes speculation costs (when employing control and data speculation etc.) and resulting compensation copies into account.

The support of long-range upward and downward code motion, different sorts of speculation, lazy compensation code insertion, and predication makes wavefront scheduling one of the most complex and powerful global scheduling techniques. An implementation for the Itanium processor achieved a 30% speedup on SPECint 95 over local scheduling (assuming perfect caches) [BMM00].

One drawback is, however, that it fully relies on path vectors, which limit the complexity of the scheduling region. Furthermore, it is a greedy, top-down algorithm: scheduling decisions are made one at a time, based on a heuristic selection function, and never reconsidered. When deciding about scheduling an instruction into a block, it is unknown whether a better opportunity will arrive in a later wavefront—this depends on other scheduling decisions still to be made. Such interdependences between decisions are well known in the area of code generation; we have already mentioned the phase-coupling problem between code generation phases.

It is evident that data dependences constitute one of the causal links between scheduling decisions that lead to interdependences. But also resource aspects play a major role: Techniques like global code motion, speculative loads, and predication *increase the demand* for execution slots (as described for code motion in Sec. 3.3). At the same, they *decrease* the schedule length (this is why they are applied) and thereby the *supply* of execution slots in the schedule. The compiler must find a trade-off between these two conflicting effects when applying such features. A too conservative use can lead to empty, wasted execution slots, contrary to the EPIC philosophy. But overuse can force the schedule length to rise again due to execution-slot shortage, which could spoil the benefit.

These multifaceted interactions are our incentive to employ integer linear programming: It has the potential to resolve all interdependences between scheduling decisions and to deliver a *global optimum*. Note that the notion of "optimality" is used in an algorithmic sense here: The resulting schedule is optimal with respect to our mathematical definition of the global scheduling problem (to be developed in the next section). In a wider context, however, optimality is relative and more complex: for example, it depends on the input set, which we must approximate by profiling information. Also there are a lot of difficult to predict, influential dynamic effects like cache misses interacting with the schedule. Clearly, no mathematical model can fully and precisely describe and minimize these runtime effects. We can achieve strict optimality only within a well-defined problem scope. However, on a statically scheduled architecture, there should be a strong correlation between schedule length and performance—this is also what our experiments in Chapter 7 confirm.

3.3.2 Formalization

The previous section has shown that numerous algorithms exist that deal with the global scheduling problem. Yet a general, formal definition of the problem is missing in the literature. Before we develop such a formalization in this section as the first step towards an ILP model, we summarize several common properties of the algorithms:

- They all divide the input program into scheduling regions and tackle them separately.
- Code motion occurs only inside these regions along control flow paths.
- Some permit only upward code motion.
- Most allow speculative code motion.
- Some feature generation of compensation code.
- Most do not schedule more than one copy of an instruction on a single path.
- Most use profiling information to estimate the profitability of code motion.
- All aim at reducing the schedule length, giving blocks with higher execution frequencies a higher priority.

Our goal is a unifying problem definition that complies with all these observations. One of the challenges is to provide a straightforward, general, and flexible formalization of when and where compensation copies are necessary. As we will see, the key to finding such a formalization is a *path-based view* of the problem. We begin with some basic definitions:

Definition 3.3.1 (Scheduling Positions, Reserved Cycles) Instructions can be scheduled at *(scheduling) positions* in the basic blocks. The set of all possible scheduling positions is given by $Pos \subseteq \mathcal{B} \times \mathbb{N}_+$. A tuple $(B, s) \in Pos$ refers to the s-th position or *(machine) cycle* s in the basic block B. An order on the positions is defined as follows:

$$(A, s_A) \prec (B, s_B) \Leftrightarrow \begin{cases} A \prec B \text{ in } G_B & \text{if } A \neq B \\ s_A < s_B & \text{if } A = B \end{cases} \tag{3.3.1}$$

Pos is a finite set: For each basic block A, a maximal number of cycles is given and denoted by $\mathbb{G}_A$. Then $(A, 1), \ldots, (A, \mathbb{G}_A)$ is the range of scheduling positions in Pos belonging to block A. $\mathbb{G}_A$ is also called the number of *reserved cycles* of A; the total number of reserved cycles is denoted by $\mathbb{G} := \sum_{A\in\mathcal{B}} \mathbb{G}_A$. □

Each scheduling position consists of one or more execution slots that take up the individual instructions. Having a finite number of these positions is a realistic assumption that leads to a finite solution space. This is useful with respect to the later transformation of the definition into an ILP model. Now we consider in which blocks instructions *may* be scheduled. We briefly recapitulate two notions from the previous sections: For each instruction $n \in V$, we call the block where it originates from before scheduling *source block*, denoted by $s(n)$. Code motion moves the instruction from this source block to a *destination block*. The *possible* destination blocks are collected in a set:

Definition 3.3.2 (Destination Block Candidates) For each instruction $n \in V$, the set $\Theta(n) \subseteq \mathcal{B}$ denotes its *destination block candidates*, i.e., those blocks where copies of n can be scheduled. □

The source block and all its predecessors and successors in the BBG are potential destination block candidates, thus we can initially assume that Θ contains all those blocks. However, the range of destination block candidates must be constrained for non-speculative instructions (see Sec. 2.1.5.1): They must not be executed speculatively, which can be ruled out if the source block *dominates* and *postdominates* the destination block for *downward* and *upward* code motion, respectively. Accordingly, we exclude from $\Theta(n)$ for a non-speculative instruction n

- all those predecessors of $s(n)$ that are not postdominated by $s(n)$ and
- all those successors of $s(n)$ that are not dominated by $s(n)$.

In the course of this thesis, we will encounter more reasons why further blocks should or must be excluded from the range of destination block candidates. Thus we make—apart from the above exclusion for non-speculative instructions—here no further assumptions on Θ and regard its definition as a given integral part of a global scheduling problem instance.

From now on, we often call destination block candidates simply "*candidate blocks*". The distinction between them and destination blocks is important: in candidate blocks copies of an instruction *may* be scheduled. If a copy *is* actually scheduled in one of these blocks, then the latter becomes a destination block of the instruction. Hence the notion “candidate block” is related to a scheduling problem instance and ”destination block” to a solution of this problem—a schedule.

Now we examine what constitutes semantics-preserving global scheduling. Before scheduling, we can regard the scheduling region as an *original* global schedule, σ_o, where all instructions are scheduled in their respective source blocks (in the order given by the control flow graph). The process of global scheduling then can be viewed as a *transformation* between global schedules that rearranges instructions, but does not change the control flow structure (although it may empty some blocks). Hence the set of *program paths*—paths that go from an entry block to an exit block through the scheduling region—remains unchanged. This allows us to take a path-based view of semantics preservation and say that a transformation from schedule σ_o to σ is correct if the same computations (and possibly exceptions) are performed in both schedules along every program path.

To be more precise, this is the case when all instructions that occur along a path in σ_o also occur along this path in σ, and when all dependences between these instructions are preserved. Additionally, *non-speculative* instructions may only appear on a path in σ if they appear along this path in σ_o, too.

These insights are embodied by the following definition: The equations (3.3.2) hold for each instruction $n \in V$ and for each path in $\mathcal{C}(s(n))$, i.e., for each path through n’s source block (see Def. 3.2.4). They ensure that in σ each such path contains exactly one copy of n, as is the case with σ_o ($\mathcal{B}(P) \subseteq \mathcal{B}$ denotes all blocks along the path P). The constraints (3.3.3) enforce that instructions are only scheduled in their candidate blocks. The choice of the candidate blocks for non-speculative instructions makes sure that they appear only on paths in σ that pass through their respective source blocks. Equ. (3.3.4) guarantees that instructions are assigned to suitable execution units.

The precedence constraints (3.3.5) preserve the data dependences. The paths in $\mathcal{C}(s(m)) \cap \mathcal{C}(s(n))$ are exactly those paths on which both m and n are located in σ_o. Then the two copies of these instructions along each such path in σ must be scheduled at positions that do not violate the dependence $(m, n) \in E_D$. Two such positions (A, s_A) and (B, s_B) respect the dependence if block A is a predecessor of B, or if both blocks are equal and n is scheduled at least w_{mn} cycles later than m there (as stated by the two rows of (3.3.5), together with Def. 3.3.1).

Remarkably, the latency w_{mn} in this definition is only taken into account *within* basic blocks. The propagation of latencies greater than one *between* basic blocks is not incorporated. Thus only intra-block stalls and no inter-block stalls due to long latencies[3] are allowed for. Since most of the Itanium 2’s latencies are zero or one cycle, this inexactness is tolerated for the sake of simplicity in both the definition and the later ILP model. Nevertheless, we will discuss possible corrective measures later in Sec. 6.7.

The resource constraints (3.3.6) finally make sure that not more instructions are scheduled at a position than can be executed per cycle by the target processor. Note that the form of these constraints is Itanium-specific in the sense that it implies that all execution units have a throughput of one operation per cycle (in other words, they are fully pipelined, see Sec. 2.2.3.3).

[3]In general, “long latencies” refers to latencies greater than one cycle .

Definition 3.3.3 (Global Schedule) Let the BBG and DDG of a scheduling region be given as defined in Def. 3.2.3 and 3.2.6. Let $s : V \longrightarrow \mathcal{B}$ give the source block of each instruction and $\Theta : V \longrightarrow \mathcal{P}(\mathcal{B})$ the set of candidate blocks. Pos is the given finite set of scheduling positions according to Def. 3.3.1. Furthermore, let $R(n) \subseteq R$ denote the set of functional unit types on which an instruction n can be executed, and let R_k denote for each type $k \in R$ the number of functional unit instances.

A *global schedule* is a mapping $\sigma : V \longrightarrow \mathcal{P}(Pos \times R)$ that assigns to each instruction a set of positions and functional unit types such that for each instruction $n \in V$ holds:

$$\forall P \in \mathcal{C}(s(n)) : \ |\{((A, s_A), k) \in \sigma(n) \,|\, A \in \mathcal{B}(P)\}| = 1 \tag{3.3.2}$$

$$\{A \in \mathcal{B} \,|\, ((A, s_A), k) \in \sigma(n)\} \subseteq \Theta(n) \tag{3.3.3}$$

$$\{k \in R \,|\, ((A, s_A), k) \in \sigma(n)\} \subseteq R(n) \tag{3.3.4}$$

Moreover, *precedence constraints* exist for each DDG edge $(m, n) \in E_D$:

$$\begin{array}{r} \forall P \in \mathcal{C}(s(m)) \cap \mathcal{C}(s(n)), \\ \forall ((A, s_A), k) \in \sigma(m) \text{ s.t. } A \in \mathcal{B}(P), \\ \forall ((B, s_B), l) \in \sigma(n) \text{ s.t. } B \in \mathcal{B}(P) : \end{array} \quad \begin{cases} (A, s_A) \prec (B, s_B) & \text{if } A \neq B \\ (A, s_A + w_{mn}) \preceq (B, s_B) & \text{if } A = B \end{cases} \tag{3.3.5}$$

Resource constraints are generated for each position and for each execution unit type:

$$\forall pos \in Pos, \forall k \in R : \ |\{n \in V \,|\, (pos, k) \in \sigma(n)\}| \leq R_k \tag{3.3.6}$$

Sometimes, a global schedule is said to be *feasible* to emphasize that it meets the above constraints. □

A consequence of this definition is that along no control flow path two or more copies of the same instruction can be scheduled:

Proposition 3.3.4 *If in a global schedule a copy of an instruction is scheduled in a block, then no further copy of the same instruction can be scheduled in this block or in one of its predecessor or successor blocks.* □

PROOF Suppose for the purpose of contradiction that two copies of an instruction n are scheduled in $A \in \Theta(n)$ and $B \in \Theta(n)$ with $A \preceq B$. We show that there exists a control flow path that traverses A, B, and $s(n)$—the contradiction then results from Equ. (3.3.2). We note that both A and B are either a predecessor or a successor of $s(n)$ (or equal to this source block) and distinguish three possible cases:

- If $A \prec s(n)$ and $B \prec s(n)$, then there exists a path leading from A to B and from B to $s(n)$.
- If $A \prec s(n)$ and $s(n) \preceq B$, then there exists a path leading from A to $s(n)$ and from $s(n)$ to B.

- If $s(n) \preceq A$ and $s(n) \preceq B$, then there exists a path leading from $s(n)$ to A and from A to B. ■

Global scheduling can be regarded as the problem to find a global schedule that yields maximal performance. In line with the examined heuristics, this is a schedule with a *minimal global schedule length*, which we define as the sum of the schedule lengths of all basic blocks, each weighted by the execution frequency of the block (given by the values $f_A \in \mathbb{R}$):

Definition 3.3.5 (Global Scheduling) *Global scheduling (GS)* is the following *minimization problem*: Given is a scheduling region together with the execution frequencies. Find a global schedule according to Def. 3.3.3 that minimizes the global schedule length:

$$\forall n \in V,\ \forall((A,s),k) \in \sigma(n):\ s \leq T_A \tag{3.3.7}$$

$$\min \sum_{A\in\mathcal{B}} f_A \cdot T_A \tag{3.3.8}$$

□

In any solution that is minimal with respect to the value of (3.3.8), T_A equals the schedule length of block A. In other words, in any optimal solution, T_A is equal to the latest cycle where an instruction is scheduled in A:

$$T_A = \max\{s \in \mathbb{N}_+\,|n \in V,\ ((A,s),k) \in \sigma(n)\}$$

The sum in (3.3.8) has then exactly the value of the global schedule length. Remarkably, the latter is only defined on the basis of a node profile (via f_A) and not on the basis of a more precise edge or path profile. This is in our case sufficient since we ignore inter-block stalls. Under these circumstances, a path profile version of (3.3.8) would yield the same results as the node profile version. Besides, only node profiles will be available later in the experiments.

The schedule length of a basic block is often simply called its *block length* and the inequalities (3.3.7) are referred to as *block length constraints*. They do not constrain the set of feasible schedules, but the set of schedules with certain maximum block lengths (T_A). In doing so, they provide the link between the schedule constituted by σ and the objective function (3.3.8) via the T_A variables. Our later analysis will reveal that these constraints play a pivotal role in determining the complexity of global scheduling. To support the complexity analysis and the stepwise development of the ILP model, we distinguish four subproblems:

Definition 3.3.6 (PCGS) *Precedence-constrained global scheduling (PCGS)* is the global scheduling problem as defined in Def. 3.3.5, but without the resource constraints (3.3.6). □

Definition 3.3.7 (RCGS) *Resource-constrained global scheduling (RCGS)* is the global scheduling problem as defined in Def. 3.3.5, but without the precedence constraints (3.3.5). □

If we omit additionally the block length constraints, we have no longer the possibility to incorporate the global schedule length or the dynamic block lengths (i.e., the values T_A, which depend on σ) into the problem formulation. Without these minimization criteria, the problem becomes a *feasibility problem*, namely the problem to find a global schedule that fits into the static block length limitations imposed by the numbers of reserved cycles of the basic blocks[4].

[4]If we allow unlimed block lengths, a feasible schedule can evidently always be found (the original schedule).

Definition 3.3.8 (PCGS-B) *Precedence-constrained global scheduling without block length constraints (PCGS-B)* is the following feasibility problem: Given is a scheduling region as defined in Def. 3.3.3. Find a feasible global schedule in the absence of resource constraints (3.3.6). □

Definition 3.3.9 (RCGS-B) *Resource-constrained global scheduling without block length constraints (RCGS-B)* is the following feasibility problem: Given is a scheduling region as defined in Def. 3.3.3. Find a feasible global schedule in the absence of precedence constraints (3.3.5). □

Chapter 4

Integer Linear Programming

Since the invention of the simplex algorithm by George B. Dantzig over fifty years ago [Dan51], *linear programming* (LP) has developed into an indispensable tool for the formulation and solution of optimization problems. This applies especially to the substantially more powerful—and substantially more difficult to solve—*integer* linear programming (ILP) variant. Both LP and ILP minimize a linear objective function subject to linear constraints. The distinguishing feature of ILP is that the variables range over a discrete set, namely a subset of the integers, which enables the modeling of *combinatorial* or *discrete* optimization problems.

The potential of ILP was almost immediately recognized after its discovery in the fifties [BFG$^+$00]. But insufficient hardware and software have soon led to some disillusionment and to the perception that ILP has very limited practical applicability. In the last years, however, this situation has changed "dramatically" due to advances in solution algorithms as well as ILP formulations [BFG$^+$00]. This is also confirmed by our own experiences.

Typical ILP applications concern "the management and efficient use of scarce resources to increase productivity" [NW88], a goal that is pursued—in the broader sense—also by this work. *Operations research*, an interdisciplinary field involving mathematics, computer science, and economics, studies the use of combinatorial optimization for decision-making in business, industry, and government. This includes areas like supply chain planning, transportation logistics, and portfolio selection, where "large [ILP] models are routinely solved in many production applications" today [ILO03b]. Other scientific applications comprise statistics (data analysis), physics (determination of minimum energy states), bioinformatics (protein-protein docking), and electrical engineering (VLSI circuit design).

This chapter provides a brief introduction to the fundamental definitions, theorems, and algorithms of integer linear programming. It covers only aspects relevant to this thesis; more comprehensive surveys are available in text books like [NW88, NW89, Sch86].

4.1 The Theory of Linear Programming

We begin with the introduction of linear programming which constitutes the basis of the integer variant in both theory and practice. For example, we will see that every ILP can be reformulated as an LP. Beyond, ILP solvers often employ solution algorithms for linear programming as subroutines. To begin with, the following is a collection of some basic definitions and results from [NW88]:

1. A subset C of $\mathbb{R}^n$ is *convex* if $\lambda x + (1-\lambda)y$ belongs to C for all $x, y \in C$ and each $\lambda \in \mathbb{R}$ with $0 \leq \lambda \leq 1$. The *convex hull* $conv(X)$ of a set $X \subseteq \mathbb{R}^n$ is the smallest convex set containing X.

2. A subset P of $\mathbb{R}^n$ is called a *polyhedron* if there exists an $m \times n$ matrix A and a vector $b \in \mathbb{R}^m$ such that $P = \{x \in \mathbb{R}^n |\, Ax \leq b\}$. P is called a *polytope* if it is *bounded*, i.e., if there exists an $\omega \in \mathbb{R}^n$ such that $P \subseteq \{x \in \mathbb{R}^n | -\omega \leq x \leq \omega\}$. P then is the convex hull of finitely many points in $\mathbb{R}^n$.

3. The points $x_1, \ldots, x_k \in \mathbb{R}^n$ are called *linearly independent* if there do not exist $\lambda_1, \ldots, \lambda_k \in \mathbb{R}$ such that $\sum_{i=1}^k \lambda_i x_i = 0$ and such that the λ_i are not all equal to 0. They are called *affinely independent* if there do not exist $\lambda_1, \ldots, \lambda_k \in \mathbb{R}$ such that $\sum_{i=1}^k \lambda_i x_i = 0$, $\sum_{i=1}^k \lambda_i = 0$ and such that the λ_i are not all equal to 0.

4. A polyhedron $P \subseteq \mathbb{R}^n$ is of *dimension* k, denoted by $\text{dim}(P) = k$, if the maximal number of affinely independent points in P is $k+1$. It is called *full-dimensional* if $\text{dim}(P) = n$.

5. Given a vector π and a real π_0, the inequality $\pi x \leq \pi_0$ is called *valid* for a nonempty polyhedron P if it is satisfied by all points in P. In this case the polyhedron $F = \{x \in P |\, \pi x = \pi_0\}$ is called a *face* of P, and $\pi x \leq \pi_0$ is said to *represent* F. F is called *proper* if $F \neq \emptyset$ and $F \neq P$; it is a *facet* of P if $\text{dim}(F) = \text{dim}(P) - 1$. An inequality representing a facet is called *facet-inducing*.

As an example, Fig. 4.1 shows a three-dimensional polytope with three faces F_0, F_1, and F_2 of dimension 0, 1, and 2, respectively. F_2 forms a facet and it holds $F_0 \subset F_1 \subset F_2$. In general, it can be shown that a proper face is a facet if and only if it is inclusionwise maximal [NW88]. The bounded faces of dimension one are called *edges* of a polytope; each edge connects two zero-dimensional faces, which are called *extreme points*—their convex hull is equal to the polytope. We often employ a different, equivalent definition:

Definition 4.1.1 (Extreme Point) A point $x \in P$ is called an *extreme point* of P if there do not exist $w, y \in P, w \neq y$ and $c \in\,]0, 1[$ such that $cw + (1-c)y = x$. □

Given an n-dimensional polyhedron, the extreme points are exactly the intersection points of n facets whose corresponding matrix rows are linearly independent:

Theorem 4.1.2 ([Sch03b]) *Let $P = \{x \in \mathbb{R}^n |\, Ax \leq b\}$ be a polyhedron and let $z \in P$. Furthermore, let $A_z x \leq b_z$ be the system consisting of those inequalities from $Ax \leq b$ that are satisfied by z with equality. Then z is an extreme point of P if and only if* $\text{rank}(A_z) = n$. □

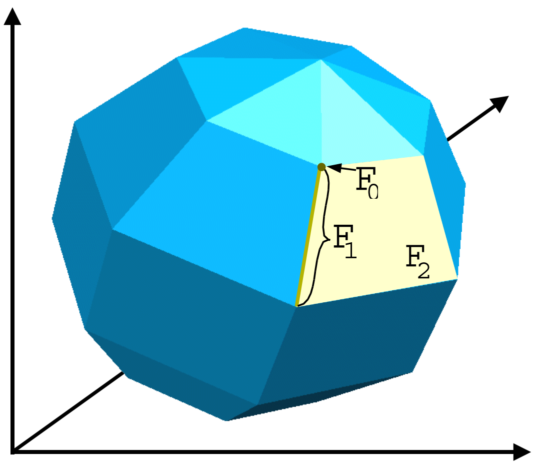


Figure 4.1: Faces of a polytope.

Each polytope P is uniquely characterized by its extreme points, or by its facets: In [NW88] it is shown that for each facet, one of its representing inequalities is *necessary* in the description of P, while every inequality that represents a face but not a facet is *irrelevant*.

Furthermore, each full-dimensional polyhedron has a unique (to within scalar multiplication) minimal representation by a finite set of linear inequalities. In particular, this representation contains for each facet exactly one inequality (unique to within scalar multiplication) representing this facet (Theorem 3.5 of [NW88]).

A polyhedron plus the vector of an objective function can be interpreted as a linear program:

Definition 4.1.3 (Linear Program) Let a *linear program* (LP) be given by a polyhedron $P_F = \left\{x \in \mathbb{R}^n_+ \,|Ax \geq b\right\}^1$, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m \times n}$. *Linear programming* is the following optimization problem:

$$\text{Minimize (or maximize) the objective function } c^T x \text{ subject to } x \in P_F$$

P_F is called the *feasible region* and each $x \in P_F$ a *feasible solution*. The LP is said to be *feasible* if $P_F \neq \emptyset$. An $x^* \in P_F$ with a minimal objective function value $c^T x^* = \inf \left\{c^T x \,|x \in P_F\right\}$ is called an *optimal solution*. If the infimum does not exist, then the problem is either *unbounded* ($P_F \neq \emptyset$) or *infeasible* ($P_F = \emptyset$). □

[1]The polyhedron is often given as $P_F = \left\{x \in \mathbb{R}^n_+ \,|Ax \leq b\right\}$. Both forms are equivalent because we can multiply the inequalities by (-1). Moreover, we often assume in the following proofs that the nonnegativity constraint $x \in \mathbb{R}^n_+$ is explicitly expressed by additional inequalities $x_i \geq 0$ ($\forall 1 \leq i \leq n$).

Example 1 The following linear program has an optimal solution with $x_1 = 1$, $x_2 = 1.5$ and an objective function value of 2.5:

$$\begin{array}{lll}
\text{min} & x_1 + x_2 & \\
\text{s. t. :} & & \\
\text{(I)} & x_1 + 2x_2 & \geq 4 \\
\text{(II)} & 3x_1 + 2x_2 & \geq 6 \\
\text{(III)} & -x_1 + 2x_2 & \geq 0
\end{array}$$

□

Intuitively, the goal of linear programming is to find a point inside a polytope that is farthest in the direction of a vector. The following theorem 4.5 from [NW88] shows that it is sufficient to consider only the extreme points for this:

Theorem 4.1.4 *Let an LP with a full-dimensional polyhedron be given. If it has an optimal solution, then there exists also an optimal solution that is an extreme point.* □

This theorem is exploited by the *simplex algorithm*, an established and efficient method to solve linear programs. Essentially, simplex works as follows: The first phase determines a feasible extreme point (illustrated in Fig. 4.2).

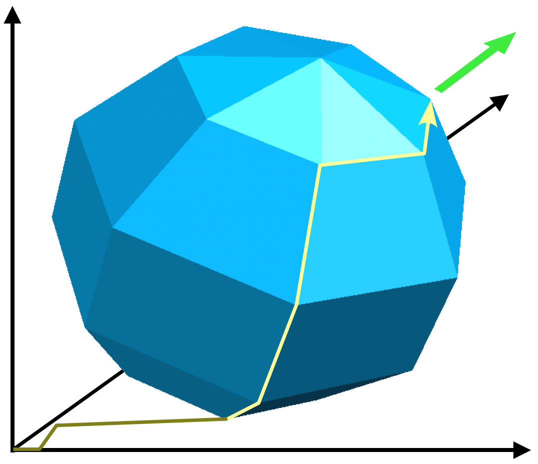


Figure 4.2: Illustration of the simplex algorithm.
The path shows the process of finding a feasible solution (darker shade) and then of an optimal solution (along the polytope edges). The upper right arrow depicts the objective function.

The second phase tries to advance iteratively to adjacent extreme points with lower objective function values by moving along the edges of the polytope (performing a *simplex step*, or *pivot step*). Candidates for this are all edges that lead in the direction of a nonincreasing objective function value. If such an edge does not exist, the present solution is known to be optimal.

The method terminates since there can be no more than $\binom{m}{n}$ extreme points in a polytope (the number of possibilities to intersect n out of m possible facets), and since cycling can be avoided. There exist problem classes where simplex actually traverses $\Omega\left(\binom{m}{n}\right)$ extreme points before it reaches an optimal one, hence its worst-case solution time is exponential.

Nevertheless, the simplex method has proven to be highly successful in the solution of real-world problem instances. This observation has been supported by probabilistic analysis, which shows that its *expected* solution time is polynomial (under rather general assumptions on the underlying distribution of instances) [NW88].

In addition, there are two polynomial-time solution methods that prove that linear programming is in $\mathcal{P}$: the ellipsoid algorithm of Khachiyan [Kha80] and the projective algorithm of Karmarkar [Kar84]. For both, the linear program must be reformulated as a feasibility (interior-point) problem. While the former is considered to be only of theoretical value, the latter is regarded as a competitive alternative to the simplex method in practice.

An important property of linear programs—also with respect to integer programming—is *duality*. A possible approach to duality is based upon the observation that linear combinations of the constraints of an LP can yield lower bounds on the objective function value. For example, the combination $\frac{1}{4} \cdot (\mathrm{I}) + \frac{1}{4} \cdot (\mathrm{II})$ of the constraints from Example 1 gives the inequality $x_1 + x_2 \geq 2.5$, which implies—as it must be satisfied by all feasible solutions—a lower bound on the objective function of 2.5. This bound is even tight in this case.

The process of finding a maximal lower bound through linear combinations can be formulated as an LP itself, where the variable vector contains the coefficients of the linear combination described above and the objective function maximizes the resulting lower bound:

Definition 4.1.5 (Dual Problem) The *dual problem*, short *dual*, of a linear program as defined in Def. 4.1.3 is written as:

$$\begin{array}{ll} \max & y^T b \\ \text{s. t.}: & y^T A \leq c \\ & y \in \mathbb{R}^m_+ \end{array}$$

The original LP from Def. 4.1.3 is called the *primal (problem)* in this context. □

Dual and primal behave symmetrically: The dual of the dual is the primal again. The following theorem with proof in [NW88] states that the dual computes a lower bound on the primal and, beyond that—as the central result of duality theory—that this lower bound is tight:

Theorem 4.1.6 (Duality Theorem) *Let a linear program and its dual be given as denoted in Def. 4.1.3 and 4.1.5. For each pair of feasible solutions of the primal and the dual holds:*

1. *Weak Duality:*
$$y^T b \leq c^T x$$

2. *Strong Duality:*
$$y^T b = c^T x \;\Leftrightarrow\; x \text{ and } y \text{ are optimal solutions}$$ □

Hence each dual feasible solution provides a lower bound on the optimal primal objective function value, a property that will be exploited when solving integer linear programs.

4.2 Integer Linear Programming

Definition 4.2.1 (Integer Linear Program) An *integer (linear) program* (ILP) is the same as a linear program, except for the feasible region, which is:

$$P_I = \left\{ x \in \mathbb{Z}_+^n \,|Ax \geq b \right\}$$

If only a subset of the variables is defined as integers, it is called a *mixed integer linear program* (MIP). □

The feasible region of an ILP consists of integer points inside a polyhedron—thus it is no longer a convex set and none of the polynomial solution methods for linear programming can be applied. In fact, integer programming is $\mathcal{NP}$-complete due to the integrality constraint. Dropping this constraint yields the underlying linear program:

Definition 4.2.2 (Relaxation) Let a minimization problem A with feasible region $X(A)$ be given. A minimization problem R with the same objective function is called *relaxation* if for the feasible region $X(R)$ holds:

$$X(A) \subset X(R)$$

For an ILP with feasible region $P_I = \left\{ x \in \mathbb{Z}_+^n \,|Ax \geq b \right\}$, its *LP-relaxation* is the LP with the same objective function and feasible region $P_R = \left\{ x \in \mathbb{R}_+^n \,|Ax \geq b \right\}$. □

Remark 4.2.3 For the optimal objective function values of a minimization problem A and its relaxation R always holds:

$$z_A \geq z_R$$ □

It follows from this (evident) remark that each optimal solution of a relaxation is also an optimal solution of the original problem *if it is feasible there*. Hence, if an integer feasible optimal solution of the LP-relaxation is found, then the corresponding ILP has been solved to optimality, too. However, the extreme points of P_R are often not integral, as in the example of Fig. 4.3.

This would be different if the polyhedron of the ILP, which encloses all of its feasible integer points, could be made equal to the convex hull of these points, i.e., equal to $P_C = conv(P_I)$. Then simplex would compute an integral optimal solution of the LP-relaxation since it always returns extreme points as optimal solutions, and hence solve the ILP. Polyhedra with this property are called integral:

Definition 4.2.4 (Integral Polytope) A nonempty polyhedron $P \subseteq \mathbb{R}^m$ is said to be *integral* if each of its nonempty faces contains an integral point. □

Corollary 4.2.5 ([NW88]) *A nonempty polyhedron $P \subseteq \mathbb{R}_+^m$ is integral if and only if all of its extreme points are integral.* □

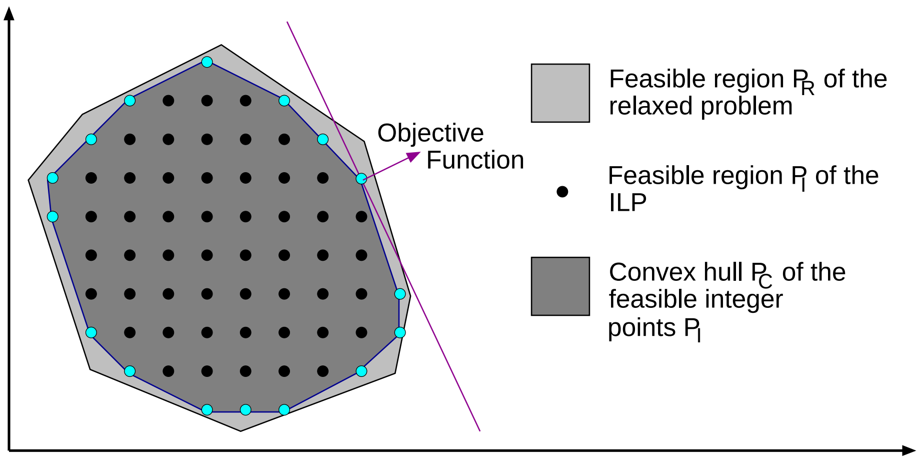

Objective Function
Feasible region P_R of the relaxed problem
Feasible region P_I of the ILP
Convex hull P_C of the feasible integer points P_I

Figure 4.3: Feasible regions of the integral and relaxed problem.

It always holds $P_I \subseteq P_C \subseteq P_R$ and in the ideal case $P_C = P_R$.

Every ILP can be reformulated such that it has a (unique) integral polytope, however, it is often unknown how and at which price: in general, there might be exponentially many constraints necessary to describe P_C [Sch86], or the computational complexity of finding these equations could be exponential—these effects would introduce the exponential complexity "through the backdoor" again. In particular, this is the case for $\mathcal{NP}$-hard problems (unless $\mathcal{P} = \mathcal{NP}$): An *efficient*—i.e., integral *and* polynomial sized—polytope of such a problem cannot exist since otherwise the ellipsoid method could solve it in polynomial time.

Although integrality often cannot be achieved in practice, it is important that the ILP model approximates P_C as close as possible. One way to achieve this is to use as many facets of P_C (called *integral facets*) in the description of P as possible. The closeness to P_C, also called *tightness* of the polytope, is crucial to the solution efficiency and often decides whether the ILP is tractable at all in practice. We will substantiate this rule theoretically in the next section and practically in Chapter 7.

4.2.1 The Branch-and-Cut Algorithm

The branch-and-bound algorithm is the standard method for solving integer programs. Many of today's ILP solvers employ an extended variant—branch-and-cut—which is outlined in the following.

The algorithm starts with solving the LP-relaxation. If the result is fractional-valued, branch-and-cut combines two different approaches to obtaining integral solutions: the first tightens the polytope by deriving and adding further constraints (*cuts*), while the second successively partitions the polytope into smaller subpolytopes and tackles them separately (*branching*).

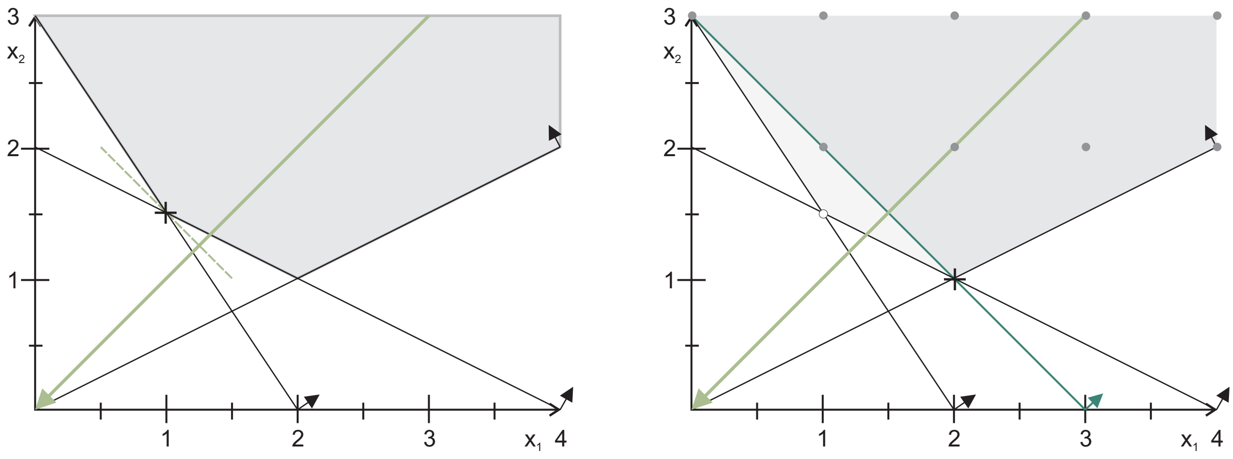


Figure 4.4: Illustration of Example 1.
The left-hand side shows the feasible region of the LP with its optimal solution (at the cross). The right-hand side shows the same problem as an ILP; only after adding the Gomory-Chvátal cut $x_1 + x_2 \geq 3$ the LP-relaxation has an integer feasible optimal solution $(2, 1)$.

A *cutting plane* (cut) is an inequality that is valid for P_I but not necessarily for P_R, i.e., that cuts away a subset of non-integral solutions. Usually, this subset includes the current (non-integral) optimal solution so that *reoptimizing* is necessary afterwards to obtain a feasible solution again. As each added inequality increases the size of the constraint matrix, cuts should be applied selectively. The process is usually guided by heuristics that estimate the effectiveness of possible cuts.

The ILP solver CPLEX automatically adds cuts from up to nine different classes, depending on the structure of the ILP [ILO03b]. The following two were most often applied to the ILPs generated in this work:

- *Clique cuts*: A group of binary variables $x_1, \ldots, x_k$ forms a clique if at most one of these variables can be non-zero in any integer feasible solution. Then the inequality $\sum_{i=1}^{k} x_i \leq 1$ is valid and under certain conditions also facet-inducing (see Sec. 4.3). This relationships can be tracked by ILP solvers via a clique table and dynamically exploited to derive cuts.

- *Gomory fractional cuts*: The idea behind these classical cuts [Gom58] can be demonstrated on the basis of the similar Gomory-Chvátal cutting planes: For any inequality $a^T x \leq \delta$ that is valid for P_R and where $a \in \mathbb{Z}^n$, the Gomory-Chvátal cutting plane $a^T x \leq \lfloor \delta \rfloor$ is valid for all integral points in P_R, and thus valid for P_I [Eis00, Sch86]. Figure 4.4 illustrates such a cut obtained from the valid inequality $x_1 + x_2 \geq 2.5$ of Example 1.

Gomory cuts are considered the most general as for any fractional-valued optimal solution of the relaxation a violated Gomory cut can be found; the successive addition of these cuts always leads to an optimal integral solution [NW88]. However, this result is, taken alone, of little value in practice since the convergence occurs much too slowly. Nevertheless, Gomory cuts have turned out to be effective if their use is integrated with the branch-and-cut process and if they are not added one at a time, but in groups [BFG$^+$00].

If the addition of cuts still does not produce an integer feasible optimal solution, a decomposition into smaller subproblems takes place which are recursively solved using the same algorithm. The solution of the whole problem is then obtained by combining the solutions of the subpolytopes as described by the following proposition:

Proposition 4.2.6 (Divide and Conquer) *Let a minimization problem be given with feasible region P and objective function $c : P \longrightarrow \mathbb{R}$, and let $P_1, \ldots, P_k$ be a partition of P, i.e., $P_i \subset P$ and $\bigcup_{i=1}^{k} P_i = P$. If we define $z_Q = \min\{c(x)\,|x \in Q\}$ for each subset $Q \subseteq P$, then*

$$z_P = \min\{z_{P_1}, \ldots, z_{P_k}\} \qquad \square$$

Since the decomposition and solution of the subpolytopes occurs recursively, a decomposition tree is formed in which each subproblem is a node. The active subproblems—those which still require further processing—are to be found in the leaves of this *branch-and-bound* tree. The partitioning usually is performed according to Dakin [Dak65]: The algorithm selects a decision variable with fractional value δ and branches on this variable by creating two subproblems with additional bounds $x_i \leq \lfloor\delta\rfloor$ and $x_i \geq \lceil\delta\rceil$, respectively. For binary variables, this means that they are fixed at 0 and 1, respectively.

Carried to the extreme, this decomposition could lead to a total enumeration of the integer feasible points, which is not viable. Hence the tree is continuously pruned, i.e., certain submodels are discarded from further processing (*fathomed*). For this purpose, the algorithm always retains the best integer solution found so far as the *incumbent (solution)*. Its objective function value, denoted by $\underline{z}$, is a monotonously decreasing *upper bound* on the objective function value of an integer feasible optimal solution. Subproblems are fathomed if they cannot contain an integral solution with an objective function value less than this bound. The incumbent is known to be globally optimal if no active subproblem remains, i.e., if for all subpolytopes P_i of the leaves of the tree

1. an optimal integral solution of P_i is known, or
2. it is known that P_i does not contain an integral solution at all, or
3. it is known that all integral solutions of P_i have an objective function value greater or equal to $\underline{z}$.

The occurrence of the second and the third case can be determined efficiently by solving the LP-relaxation of the subproblem or its dual. Regarding the second case, a subpolytope P_i does not contain an integral solution if

- its LP-relaxation is infeasible—then there can also be no integral solution,

- or if the dual of its LP-relaxation is unbounded—then it follows from duality (Theorem 4.1.6) that P_i is empty.

Regarding the third case, we observe that for the objective function value z_I of any integer feasible solution in P_i it holds $z_I \geq \underline{z}$ if

- the LP-relaxation has an objective function value $z_R \geq \underline{z}$—then it follows with Remark 4.2.3 $z_I \geq z_R \geq \underline{z}$,
- or if the dual of its LP-relaxation has a feasible solution $z_D \geq \underline{z}$—then it follows from duality $z_I \geq z_R \geq z_D \geq \underline{z}$.

The latter point gives one reason why branch-and-bound implementations usually solve the dual of the LP-relaxation at each node (or, equivalently, apply the dual simplex algorithm directly to the primal): While only an *optimal* solution of the primal delivers a lower bound, every *feasible* solution of the dual already does the same. The greatest lower bound obtained this way is often referred to as the "best bound" of the node. This value is monotonously increasing with the depth of the nodes in the tree.

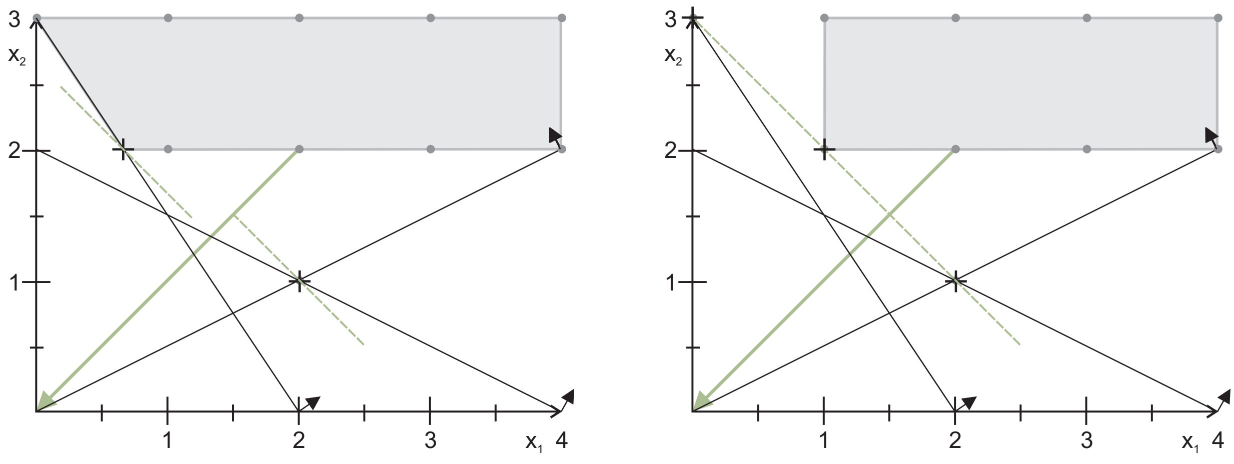


Figure 4.5: Application of branch-and-bound to Example 1.

Figure 4.5 depicts an example for a simple branch-and-bound procedure. It starts with an optimal solution of the LP-relaxation as shown in the left-hand side of Fig. 4.4 (intentionally no cuts are added). The first partitioning is $x_2 \leq \lfloor 1.5 \rfloor$ and $x_2 \geq \lceil 1.5 \rceil$, yielding the integer feasible solution $(2, 1)$ with objective function value 3 in the first of the two subpolytopes (left-hand side of Fig. 4.5). However, the best bound $\frac{8}{3}$ of the other subpolytope does not allow to exclude the possibility that a better integer feasible solution can still be found here[2]. Thus a

[2]At least for the described algorithm. A more sophisticated solver could observe that the objective function has only integral coefficients and that consequently the objective function value of every integer feasible solution is integral. This allows to round the best bound up to 3, to fathom the subproblem and to abort the branch-and-bound process already here.

further decomposition of this subpolytope occurs ($x_1 \leq \left\lfloor \frac{2}{3} \right\rfloor$ and $x_1 \geq \left\lceil \frac{2}{3} \right\rceil$, right-hand side), yielding the best bound 3 in both resulting subproblems. Hence $(2, 1)$ is an optimal solution of the ILP. The corresponding branch-and-bound tree is depicted in Fig. 4.6.

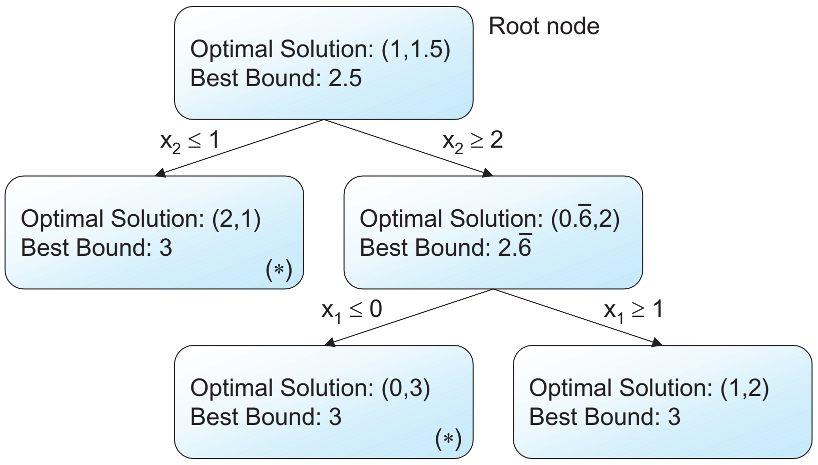


Figure 4.6: Branch-and-bound tree corresponding to Fig. 4.5.

Nodes marked with (*) have subpolytopes consisting only of a single integer vector.

From the description of the branch-and-bound process it becomes clear why a tight ILP formulation is so crucial to efficient solvability: Firstly, then more of its extreme points are integral, or have more integral components. Secondly, the bound from the LP-relaxation gets tighter and allows to prune the branch-and-bound tree earlier. This helps to contain its growth, which can be exponential in the worst case.

Besides the tightness, two major sorts of decisions during the branch-and-bound process strongly influence this growth. The **first decision** concerns the choice of the next active node for branching. After processing a node, the ILP solver can

- continue to dive deeper into the tree from there by selecting one of the two newly created successors of this node—if available—for decomposition (*depth-first search*),
- or select one of the other remaining active nodes and proceed there (*backtracking*).

The first choice prioritizes the early finding of integer feasible solutions, which are more likely to be found deep in the tree than at nodes near the root [NW88, ILO03b]. The second works at moving the best bounds, which is essential to proving that an attained incumbent is optimal, i.e., that no better integer solution can be found in another active subproblem. ILP solvers balance these two sometimes-competing aims heuristically. Descriptions of backtracking strategies can be found in [NW88, ILO03b]; they can be parameterized by the user to emphasize either early feasibility or proving optimality.

During backtracking, a subproblem has to be selected among all active leaf nodes. The most common strategy chooses the node with the lowest *best bound*, which is usually one near the top of the tree. In contrast to this, an alternative strategy chooses a node that is estimated most likely to contain an optimal solution (*best estimate*), which generally means diving deeper into the tree from a different leaf.

The **second decision** concerns the selection of a fractional-valued variable for branching at a node. It is preferable to choose variables whose integrality would also implicate the integrality of other variables. However, robust methods for identifying such variables have not been established, therefore it is common to let the user specify priorities [NW88]. [ILO03b] recommends that, if integer variables represent different types of decisions, and if a decision depends on another, then variables representing the depending decision should have a lower priority than those representing the other.

If no priority order is available or if there is more than one variable with maximal priority, then there exist a variety of heuristics for the choice of a variable; several are described in detail in [NW88, ILO03b]. It is also possible to predetermine the *branch direction* of a variable, i.e., which of the two subproblems resulting from the division should be explored first.

In modern ILP solvers, the branch-and-cut process is supported by *preprocessing* and *node heuristics*. Preprocessing applies several reductions to the ILP in order to simplify constraints, reduce the problem size, and eliminate redundancy. This can include tightening bounds, fixing and substituting variables, and removing redundant constraints [NW88, ILO03b]. In CPLEX, these operations are applied iteratively in several passes to the root node prior to solving it and—in a restricted way—also to the subproblems within the tree (*node presolve*). They can greatly facilitate the solution process (see Sec. 7.3.1, [NW88]).

Node heuristics try to find integer feasible solutions at nodes during the branch-and-cut procedure heuristically. These periodically applied routines usually start from an available feasible solution and fix successively fractional-valued variables to integers, subject to the condition that the vector remains feasible and has an objective function value less than the incumbent's one. If an integral vector results, a new incumbent has been found and the node heuristics was successful. Since successively better incumbents are highly important to move the upper bound and prune the tree through this, the computational effort of the node heuristics pays even if it succeeds only in a small fraction of all cases.

4.3 Solution Efficiency and Integral Polytopes

Although presolve tries to detect and remove redundant constraints, it can be advisable to deal with redundant constraints already during the formulation of the ILP: Some models turn out to be of polynomial size only after an exponential number of redundant constraints have been removed—we will encounter an example for this in Chapter 5. Hence we will study redundancy here in more detail:

Definition 4.3.1 (Redundancy) Given a system of linear constraints $\mathcal{H}$, a constraint $h \in \mathcal{H}$ is *redundant* in this system if there exists no point that satisfies all other constraints in $\mathcal{H} \setminus \{h\}$ but not h. □

The following lemma concretizes this definition with help of the duality theorem:

Lemma 4.3.2 (Redundancy and Subsumption) *Let a polyhedron $P = \{x \in \mathbb{R}^n | \, Ax \le b\}$ be given, and let $P' = \{x \in \mathbb{R}^n | \, A'x \le b'\}$ be the polyhedron P without the i-th constraint $h_i = (A_i x \le b_i)$. This constraint is redundant in the description of P if and only if there exists a linear combination of other constraints that* subsumes *h_i, i.e., there exists a $\lambda \in \mathbb{R}_+^{m-1}$ such that $A_i \le \lambda^T A'$ and $\lambda^T b' \le b_i$.* □

PROOF h_i is redundant in the description of P according to Def. 4.3.1 if and only if the linear program with feasible region P' and objective function max $A_i x$ has an optimal solution with objective function value less than or equal to b_i. With Theorem 4.1.6, this is the case if and only if the dual of this LP has an optimal solution $\lambda \in \mathbb{R}_+^{m-1}$ that satisfies its constraints $\lambda^T A' \ge A_i$ and has an objective function value $\lambda^T b'$ less than or equal to b_i. The existence of such an λ was to be shown. ■

This lemma provides a means for detecting and removing a redundant constraint, however, it should be used with care when applied to a *group* of constraints: The removal of a single redundant constraint can render another, previously redundant constraint irredundant. For instance, this is the case for the first two of the following inequalities, which are both redundant but cannot be both removed:

$$\begin{array}{rcll} 2x_1 + x_2 & \le & 1 & \quad (A) \\ x_1 + 2x_2 & \le & 1 & \quad (B) \\ x_1 - x_2 & \le & 0 & \quad (C) \\ -x_1 + x_2 & \le & 0 & \quad (D) \end{array}$$

The point is that for each of these two constraints, the other appears in the linear combination that constitutes its redundancy ($(A) + (D)$ for (B) and $(B) + (C)$ for (A)). Thus, when one is removed, the other is no longer redundant. The following lemma shows a way how to deal with such circular dependences:

Lemma 4.3.3 *A subset $\dot{\mathcal{H}} \subseteq \mathcal{H}$ of linear constraints is redundant in $\mathcal{H}$ if there exist*

- *a strict order $\ll$ on $\dot{\mathcal{H}}$ and*
- *for each $h \in \dot{\mathcal{H}}$ a linear combination of other constraints $h_1, \dots h_k \in \mathcal{H}$ that subsumes h and with the property $h_i \ll h$ for all $h_i \in \dot{\mathcal{H}}$.* □

PROOF We can safely apply Lemma 4.3.2 to remove all constraints in $\dot{\mathcal{H}}$ if this is done successively in decreasing order with respect to $\ll$. ■

Besides a minimal ILP size, a further goal during the formulation of the ILP is the integrality of the polytope—or at least of a subpolytope. One well-known property that implies integrality of a polytope is total unimodularity of the coefficient matrix:

Definition 4.3.4 (Total Unimodularity) An integral matrix A is *totally unimodular* if the determinant of each square submatrix of A is equal to 0, 1, or -1. □

Theorem 4.3.5 (Hoffman-Kruskal) *The polyhedron is $P = \left\{ x \in \mathbb{R}^n_+ \middle| Ax \leq b \right\}$ is integral for all $b \in \mathbb{Z}^m$ for which is it not empty if and only if A is totally unimodular.* □

Note that this theorem with proof in [NW88] does not say that each integral polytope has a totally unimodular coefficient matrix—the requirements for the forward implication are stronger, namely integrality for different vectors $b \in \mathbb{Z}^m$. Thus total unimodularity apparently characterizes only a subclass of all integral polytopes.

In fact, we will introduce now a further class of integral polytopes that goes beyond this subclass. This class is especially suited for optimization problems that model logical implications with binary variables: Many of these problems can be formulated—at least in parts—as *node packing problems* on a graph. For these problems, integral facets and, under certain conditions, entire integral polytopes are known:

Definition 4.3.6 (Node Packing Problem [Sch03b]) Let $G = (V, E)$ be an undirected graph.

- A *node packing* on G is a $U \subseteq V$ with the property that no pair of nodes in U is joined by an edge. U is also called a *stable set*.
- The *clique matrix* K_G of G is the (0,1) incidence matrix whose rows correspond to all the maximal cliques of G and whose columns correspond to the nodes of G.
- The *fractional node-packing polytope* of G is $P = \left\{ x \in \mathbb{R}^{|V|}_+ \middle| K_G x \leq 1 \right\}$. □

Theorem 4.3.7 (Padberg) *Each inequality of the fractional node-packing polytope represents an integral facet.* □

This result from [Pad73] is not sufficient to reason that the fractional node-packing polytope is integral—for this result, it had to contain *all* the integral facets, i.e., all the facets of P_C, the convex hull of its enclosed integer points. This is only the case for a special class of graphs:

Theorem 4.3.8 ([NW88]) *A graph is perfect if and only if its fractional node-packing polytope is integral.* □

This theorem reveals—in perhaps surprising clearness—that the theory of perfect graphs has deep connections to integer linear programming. There exist two major characterizations of perfect graphs, which have only recently been shown to be equivalent through the proof of the strong perfect graph conjecture [Sch03a]. We do not expand on the complex topic of perfect graphs here, but we note the following theorem from [Sch03a] for later use:

Definition 4.3.9 A graph $G = \{V, E\}$ is called *transitively orientable* (or equivalently, *comparability graph*) if each edge can be assigned a one-way direction in such a way that the resulting oriented graph (V, F) satisfies the following property:

$$(a, b) \in F \wedge (b, c) \in F \Rightarrow (a, c) \in F$$ □

Theorem 4.3.10 *Transitively orientable graphs are perfect.* □

Often we cannot apply the preceding theorems directly to the polytopes whose integrality we want to prove. Instead, we show that similar polytopes are integral and apply *integrality-preserving* transformations to obtain the actual polytope. For example, the resulting polytope remains integral if we

- turn several inequalities into equations by changing the "$\leq$" relation symbol to "$=$",
- or intersect it with a lower-dimensional plane spanned by coordinate axes,
- or project it onto a lower-dimensional plane spanned by coordinate axes,
- or duplicate columns of the constraint matrix.

The first statement, also expressed by the following lemma, is intuitively clear: if an inequality is turned into an equation, then the resulting polytope, if nonempty, is equal to the face represented by this inequality—and each face of an integral polytope is also an integral polytope. A formal proof is given in Appendix B.1.1.

Lemma 4.3.11 *Let $P = \left\{ x \in \mathbb{R}^n_+ \middle| Ax \leq b \right\}$ be an integral polyhedron. If the row indices $I = \{1, \ldots, m\}$ of $A \in \mathbb{R}^{m \times n}$ are partitioned into two subsets I_1 and I_2, then*

$$P' = \left\{ x \in \mathbb{R}^n_+ \middle| A_{I_1} x \leq b_{I_1} \wedge A_{I_2} x = b_{I_2} \right\}$$

is integral (if nonempty). □

If we apply this lemma to some of the implicit inequalities $x_i \geq 0$ that hold for all variables, then we can conclude that the polytope remains integral if we fix some variables to zero (the second statement):

Corollary 4.3.12 *Let $P = \left\{ x \in \mathbb{R}^n_+ \middle| Ax \leq b \right\}$ be an integral polyhedron. If some of the columns of A are removed from the formulation with their corresponding variables, then the resulting polyhedron remains integral (if nonempty).* □

(Formal proof in Appendix B.1.2)

The transformation of the above corollary can be regarded as an intersection with the subspace $H_J = \left\{ x \in \mathbb{R}^n_+ \middle| \forall i \in J : x_i = 0 \right\}$ for a subset $J \subseteq \{1, \ldots, n\}$, i.e., the subspace spanned by the coordinate axes of the variables x_i for all $i \notin J$. We will finally show that not only an intersection with H_J, but also an orthogonal projection onto this plane preserves the integrality of a polytope (the third statement). The projection $proj_J(x)$ of a point $x \in \mathbb{R}^n_+$ onto H_J is defined as a vector $y \in H_J$ such that

$$y_i = \begin{cases} 0 & \text{if } i \in J \\ x_i & \text{else} \end{cases}$$

The projection of a polytope is then the union of its projected points:

$$proj_J(P) = \bigcup_{x \in P} proj_J(x)$$

In particular, the projection of a polytope is the convex hull of its projected extreme points. Together with the fact that the defined kind of projection does preserve the integrality of points, the following lemma with proof in Appendix B.1.3 is then perspicuous:

Lemma 4.3.13 *If $P = \left\{ x \in \mathbb{R}^n_+ \,\middle|\, Ax \leq b \right\}$ is an integral polytope, then so is $proj_J(P)$.* □

Finally, the last lemma—also with proof in the appendix—says that duplicating columns of the constraint matrix preserves the integrality of the polytope. This transformation can be regarded as replacing variables by a sum of new variables in the ILP.

Lemma 4.3.14 *Let $P = \left\{ x \in \mathbb{R}^n_+ \,\middle|\, Ax \leq b \right\}$ be an integral polyhedron and let the $m \times n'$ matrix A' be the matrix A with some columns duplicated. Then $P' = \left\{ x \in \mathbb{R}^{n'}_+ \,\middle|\, A'x \leq b \right\}$ is integral.* □

Chapter 5

An ILP Model for Global Instruction Scheduling

After the basics of global instruction scheduling and efficient integer linear programming have been introduced in Chapters 3 and 4, respectively, this chapter will present our ILP formulation for global instruction scheduling. We will begin with formulations for two subproblems from Chapter 3: PCGS and RCGS without the resource and precedence constraints, respectively.

We aim at obtaining integral and polynomial sized polytopes for these subproblems—however, there is no general way, no silver bullet to achieve this: The determination of integral facets of a problem is considered "more of an art than a formal methodology" [NW88]. In fact, we found the presented formulation only to a lesser extent through polyhedral analyses, but more so through experience and experimentation. However, later polyhedral analyses revealed that the formulation—with some small changes—measures up to high levels of integrality, as it had been indicated by the experiments.

During the analysis, we approach the question of efficiency also from the opposite side: By identifying $\mathcal{NP}$-complete subproblems we can determine the limits of efficiency, since we know that we cannot obtain integral and polynomial sized subpolytopes for them unless $\mathcal{P} = \mathcal{NP}$. Hence, in this chapter we will prove not only that our formulation is efficient, but also that it is at the boundary of this limit imposed by $\mathcal{NP}$-hardness.

5.1 Precedence-Constrained Global Scheduling

We first develop a polytope for PCGS of Chapter 3. We briefly recapitulate the notation developed there: The basic block graph is given by $G_B = (\mathcal{B}, E_B, \mathcal{B}_{entry}, \mathcal{B}_{exit})$. $\mathcal{C}$ denotes the set of all control flow paths in G_B from an entry node (from $\mathcal{B}_{entry}$) to an exit node (from $\mathcal{B}_{exit}$) and $\mathcal{C}(A) \subseteq \mathcal{C}$ the subset of those paths that pass through block A. The set $\mathcal{B}(C)$ contains all blocks on a path $C \in \mathcal{C}$.

We say that a block A is a *predecessor* of block B if there exists a nonempty control flow subpath from A to B, also written as $A \prec B$. A is called a *direct predecessor* if there exists a BBG edge $(A, B) \in E_B$. The predecessor relationship $\preceq$ imposes a partial order on $\mathcal{B}$. The sets of A with its predecessors and successors are denoted by $\mathcal{B}^{\preceq}(A) := \{B \in \mathcal{B} \,|\, B \preceq A\}$ and $\mathcal{B}^{\succeq}(A) := \{B \in \mathcal{B} \,|\, B \succeq A\}$, respectively.

The instructions with their global data dependences are given by the acyclic data dependence graph $G_D = (V, E_D)$. Each edge $(m, n) \in E_D$ has a *latency* w_{mn} associated with it. An instruction $n \in V$ can be scheduled at a cycle $t \in G(A) := \{1, \ldots, \mathbb{G}_A\}$ in one of its candidate blocks $A \in \Theta(n) \subseteq \mathcal{B}$. This *(scheduling) position* can also be represented by a tuple (A, t) from the set $Pos \subseteq \mathcal{B} \times \mathbb{N}_+$.

In order to simplify the development of the ILP model and the accompanying proofs, we assume that the given scheduling region has several properties:

Remark 5.1.1 We make the following assumptions about the given scheduling problem and the ILP model to be developed:

1. We allow *unlimited code motion*, which means that each instruction can be moved into all predecessors and successors of its source block:

$$\forall n \in V :\ \Theta(n) := \mathcal{B}^{\preceq}(s(n)) \cup \mathcal{B}^{\succeq}(s(n))$$

   The following equation is a consequence of this definition:

$$\forall (m, n) \in E_D :\ \Theta(m) \cap \Theta(n) = \{A \in \mathcal{B}(C) \,|\, C \in \mathcal{C}(s(n)) \cap \mathcal{C}(s(m))\}$$

2. We assume that the number of reserved cycles of each block is at least two, and greater or equal to one plus the maximal latency between two dependent instructions that can be scheduled there:

$$\forall A \in \mathcal{B} :\ \mathbb{G}_A \geq 2 \wedge \mathbb{G}_A \geq \max\{1 + w_{mn} \,|(m, n) \in E_D \wedge A \in \Theta(m) \wedge A \in \Theta(n)\}$$

3. We assume that there are no JS edges. As described in Sec. 3.2, this can be achieved by adding new JS blocks.

4. We assume that there is a single, empty exit block, denoted by Ω:

$$\mathcal{B}_{exit} = \{\Omega\}$$

   This new block is added as the common direct successor of all exit blocks. It is an empty block with $\mathbb{G}_A = 0$ (the only exception from (2)).

5. We tolerate initially a small deficiency of the ILP model: We allow that a schedule may be feasible although it violates the data dependences of instructions that are scheduled in the first cycle of a basic block. □

These assumptions are not realistic and are only imposed for proof-technical reasons. They increase the number of modeled scheduling positions and consequently the size of the ILP. More seriously, (1) and (5) also affect the correctness. Therefore we will *revert* them completely after the ILP model has been developed and after its integrality properties have been proven. This will be done simply by removing variables from the formulation (i.e., removing the respective matrix columns). With respect to (5), for example, this means that we remove the possibility to schedule instructions at the first cycle of blocks—the second cycle becomes the new first cycle et cetera. According to Corollary 4.3.12, the removal of matrix columns does not compromise the integrality of a polytope. A detailed description of the reversal process follows in Sec. 5.1.2.

5.1.1 Deriving an Integral Subpolytope of PCGS-B

Initially, we focus on PCGS-B, the version of PCGS without the block length constraints. Instead of presenting the final polytope right now, we will start from a naïve node-packing formulation of the problem and improve—in a detailed, continuous process—its efficiency and complexity properties. We will do so by providing a series of incrementally improved polytopes, accompanied by formal correctness and integrality proofs. This stepwise presentation simplifies the proofs and makes it easier to adopt the employed methods to similar problems.

Our approach builds on earlier, proven ILP models for instruction scheduling, namely the seminal OASIC formulation [GE93, CWM94]: Like there, the basic idea is to first model scheduling as a node packing problem and then to exploit well-known results about integral facets and polytopes of this problem class (as discussed in Sec. 4.3). The first part of the construction uses a similar approach like [CWM94]. However, we are able to prove stronger results than [CWM94] (and correct some issues), but above all, we tackle global instead of local scheduling.

We will construct a *constraint graph* G_c such that PCGS-B corresponds to the node packing problem on this graph. Two other directed graphs are employed in order to develop G_c: We first construct a *skeleton digraph* G_s and its transitive closure G_s^*. G_c is then defined as the underlying undirected graph of G_s^* (i.e., the graph obtained by ignoring the orientation of the edges): The single purpose of this indirect construction is that it will allow us later to apply directly Theorem 4.3.10 to conclude that G_c is perfect.

$$\begin{tikzcd} \underbrace{G_s}_{\text{Skeleton Digraph}} \arrow[r, "\text{Transitive Closure}"] & \underbrace{G_s^*}_{\text{Transitively Oriented Graph}} \arrow[r, "\text{Underlying Undirected Graph}"] & \underbrace{G_c}_{\text{Perfect Constraint Graph}} \end{tikzcd}$$

The skeleton digraph $G_s = (N, E_s)$ has nodes $N \subseteq V \times Pos$, i.e., the nodes are tuples composed of an instruction and a scheduling position (sometimes written as triples with the position split up into its basic block and cycle components). We say that a node $(m, B, s) \in N$ *belongs* to instruction m, block B, and cycle s. Each node represents the decision to schedule the instruction at this position. The graph contains for all instructions $m \in V$, all candidate blocks $B \in \Theta(m)$, and all cycles $s \in G(B)$ such a node (m, B, s), plus edges from this node to the nodes (*if existing*):

- (m, A, t) if $(A = B \wedge t = s - 1)$ or $((A, B) \in E_B \wedge s = 1 \wedge t = \mathbb{G}_A)$
  (called *type 1 edge*)

- $(n, B, s + w_{mn} - 1)$ if $(m, n) \in E_D$ (called *type 2 edge*)

The type 1 edges always go between nodes belonging to the *same instruction*, namely from each node "upwards" to the node with the next lower cycle component (if the latter does not exist, edges to the last nodes of all direct predecessor blocks are added). In contrast, type 2 edges exist only between nodes belonging to the *same basic block*, namely between nodes of two data dependent instructions.

As an example, Figure 5.1 (a) depicts a simple basic block graph. The circles inside each block represent the associated nodes of G_s for two dependent instructions m and n (in this example, only one or two cycles are reserved in the basic blocks). Figure 5.1 (b) shows the skeleton digraph and (c) its transitive closure. The type 2 edges, shown in a lighter grey, are vertical in this example since we assume that the latency w_{mn} is one. Broadly speaking, the rationale behind this structure is that two nodes are connected by an edge if they represent scheduling decisions that cannot coexist in any feasible schedule.

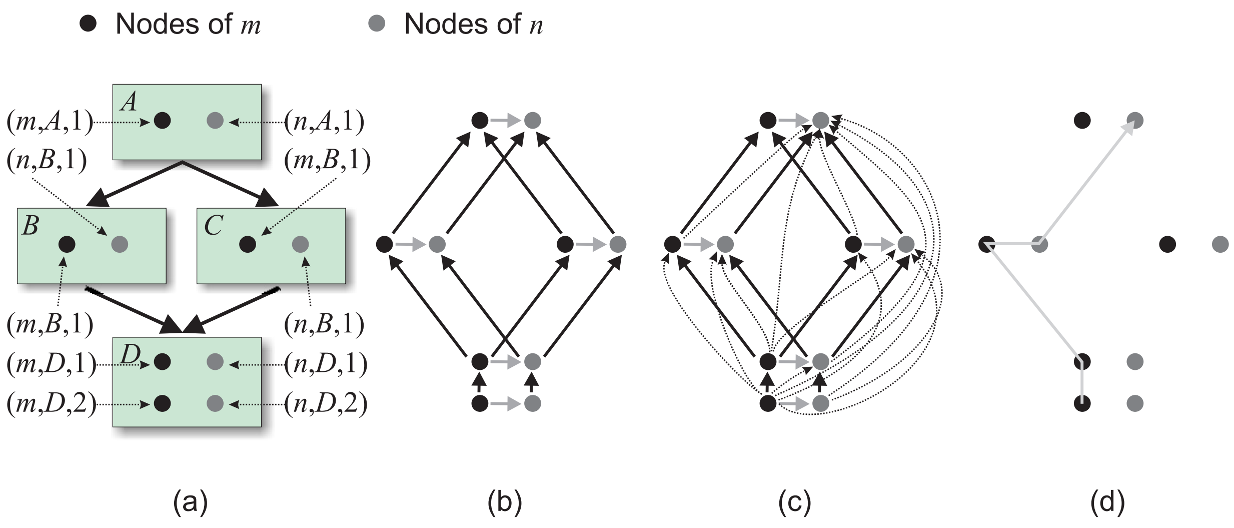


Figure 5.1: An example with four basic blocks and two instructions (a), the corresponding skeleton digraph G_s (b), its transitive closure (c), and a maximal path in G_s (d).

If an instruction n is dependent on m with latency zero, then there is no type 2 edge due to this dependence outgoing from a node of the form $(m, B, 1)$, that is, where m is scheduled in the first cycle of a block. Such an edge would go to the node $(n, B, 0)$, which does not exist (note the caveat "*if existing*" in the above definition). This absence causes the deficiency mentioned in Remark 5.1.1-(5). It could be fixed by adding type 2 edges to all direct predecessor blocks, however, this is not done for the sake of simplicity, namely in order to uphold the rule that type 2 edges exist only between nodes belonging to the same block. From now on we denote by PCGS-B* the variant of PCGS-B that tolerates this deficiency.

In the following we will often use abbreviations to describe groups of nodes with a particular component, that is, we denote by

- $N(m) = \{(n, B, s) \in N \,|n = m\}$ those nodes that belong to instruction m

- $N(A) = \{(n, B, s) \in N \,|B = A\}$ those nodes that belong to block A
- $N(C) = \bigcup_{A \in \mathcal{B}(C)} N(A)$ the nodes of all blocks along the control flow path C
- $N(P)$ those nodes on a path $P \subseteq G_s$

As it can be seen from the last definition, we describe a path in G_s often as a subgraph, written as $P \subseteq G_s$. Of special interest are the *maximal* paths in G_s, i.e., those which cannot be extended by appending edges to their first or last node (see Figure 5.1 (d)). This is also expressed by the following definition:

Definition 5.1.2 (Maximal Subgraphs) Given a graph G, a clique (or a path, tournament) $F \subseteq G$ is called *maximal* if there exists no other clique (path, tournament) $H \subseteq G$ such that $F \subset H \subseteq G$. □

The aforementioned tournament can be regarded as a complete directed graph:

Definition 5.1.3 (Tournament) A *tournament* is a directed graph in which each pair of distinct nodes $\{u, v\}$ is joined by exactly one edge (u, v) or (v, u). □

Before we proceed with the formulation of the node packing problem, we note for later use three facts that arise from the construction of G_s:

Proposition 5.1.4 *For every maximal path $P \subseteq G_s$, there exists a unique control flow path $C \in \mathcal{C}$ such that $N(P) \subseteq N(C)$.* □

PROOF While type 2 edges go only between nodes belonging to the same block, type 1 edges can also go between two nodes belonging to two distinct blocks, but only if they are connected by a control flow edge. We traverse P in the opposite direction and concatenate these control flow edges—the result is a unique control flow path.

It remains to be shown that this path is complete, i.e., that it goes from an entry to an exit block. We remember that we have allowed unlimited code motion (Remark 5.1.1-(1))—as a result, the first node of every *maximal* path in G_s must belong to the last cycle of an exit block, and the last node to the first cycle of an entry block (otherwise it could be extended to include such nodes by using type 1 edges). Hence the obtained control-flow path goes from an entry to an exit block. ■

Proposition 5.1.5 *Let (a, b) be a type 1 edge, then there exists no other path from a to b in G_s consisting of type 1 edges.* □

PROOF Let us assume that such a path $(a, c, \ldots, b)$ exists. Let the nodes b and c belong to the basic blocks B and C, respectively. Then B and C must be two distinct direct predecessors of A since there are two outgoing type 1 edges at node a so that the cycle component of a must be one—then from the construction of G_s it follows $B \neq C$, $A \neq B$ and $A \neq C$. In addition, since we have a path from c to b consisting of type 1 edges, it follows $B \preceq C$ and (B, A) is a JS edge—which contradicts the assumption that the BBG has no JS edges (Remark 5.1.1-(3)). ■

Proposition 5.1.6 *If a type 2 edge connects two nodes* $(m, A, s) \in N(A)$ *and* $(n, A, s + w_{mn} - 1) \in N(A)$*, then there exists a path* $C \in \mathcal{C}(s(m)) \cap \mathcal{C}(s(n))$ *that passes through* A*.* □

PROOF The existence of the type 2 edge implies $A \in \Theta(m)$ and $A \in \Theta(n)$. Thus A is either a predecessor or a successor of $s(n)$ and $s(m)$ (or equal to one or both of these source blocks). Since there exists a DDG edge from m to n, there also exists a (possibly empty) control flow subpath from $s(m)$ to $s(n)$. The latter can be used to connect A, $s(m)$ and $s(n)$ via a control flow path in the following way:

- If $A \prec s(n)$ and $A \prec s(m)$, then there exists a path leading from A to $s(m)$ and from $s(m)$ to $s(n)$.
- If $s(m) \preceq A$ and $A \prec s(n)$, then there exists a path leading from $s(m)$ to A and from A to $s(n)$.
- If $s(m) \preceq A$ and $s(n) \preceq A$, then there exists a path leading from $s(m)$ to $s(n)$ and from $s(n)$ to A.

This subpath can be extended to a complete path C with the desired properties. ■

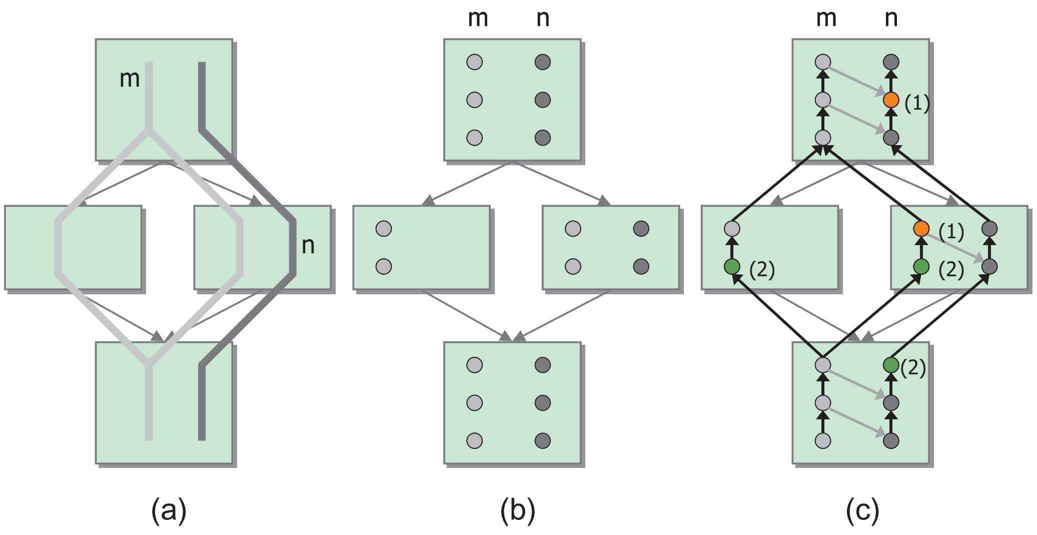


Figure 5.2: Two data dependent instructions m and n ($w_{mn} = 2$) with their candidate block ranges (a), the corresponding nodes (b), and the graph G_s with an infeasible (1) and a feasible (2) solution (c). The two nodes of (1) do not constitute a node packing since they are connected by a path in G_s—they represent the decision to schedule n before m, violating the dependence.

Now we are ready to define the first polytope of PCGS-B*. The following theorem states that the node packings on G_c—which is, as mentioned earlier, the underlying undirected graph of the transitive closure G_s^* of G_s—correspond exactly to the schedules of PCGS-B* (under the condition that a further class of inequalities is added as in the theorem below). The hereby intended relationship between node packings and schedules is obvious: A node (m, B, s) is element of a node packing if and only if a copy of instruction m is scheduled at cycle s in block B in the corresponding schedule (see also the example in Fig. 5.2). In the theorem, we directly give a naïve integer linear programming formulation of the node packing problem: We employ for each node a in G_c (which has the same nodes as G_s and G_s^*) a binary variable x_a that is one if and only if the node is contained in the node packing.

The purpose of Equ. (5.1.2) is to ensure that instructions *are* scheduled, namely once along each program path through their respective source blocks. Otherwise the empty set would—as a trivial node packing—also be a solution. Equ. (5.1.2) model the *requirement* of certain scheduling decisions, as opposed to (5.1.1), which models the *exclusion* of incompatible decisions.

Theorem 5.1.7 (PCGS-B* Polytope I) *The following inequalities form a polytope of PCGS-B*:*

$$\forall \textit{ edges } \{a, b\} \textit{ in } G_c : \; x_a + x_b \leq 1 \tag{5.1.1}$$

$$\forall n \in V, \forall \textit{ paths } C \in \mathcal{C}(s(n)) : \sum_{a \in N(C) \cap N(n)} x_a \geq 1 \tag{5.1.2}$$

□

Proof We have to show that the above description of PCGS-B* is equivalent to Def. 3.3.3 without the resource constraints (3.3.6). Let there be an assignment of the x variables and a mapping σ given such that both formalisms describe *the same schedule*:

$$x_{(n,A,s)} = 1 \Leftrightarrow (A, s) \in \sigma(n) \quad \forall n \in V, \forall A \in \mathcal{B}, \forall s \in G(A)$$

Here $\sigma : V \longrightarrow \mathcal{P}(Pos)$ is defined differently than in Def. 3.3.3 since we omit the resource binding. It is clear that σ satisfies (3.3.3) since the existence of a variable $x_{(n,A,s)}$ implies $A \in \Theta(n)$. It remains to be shown that the variable assignment satisfies (5.1.1) and (5.1.2) if and only if σ satisfies (3.3.2) and (3.3.5)—modulo the deficiency described in Remark 5.1.1-(5). This statement can be written as

$$\mathcal{X}_1^\alpha \wedge \mathcal{X}_1^\beta \wedge \mathcal{X}_2 \Leftrightarrow \mathcal{Y}_1 \wedge \mathcal{Y}_2 \tag{5.1.3}$$

where:

- $\mathcal{X}_1^\alpha \Leftrightarrow$ The variable assignment satisfies (5.1.1)-α, which denotes all those instances of (5.1.1) where a and b are nodes that belong to the same instruction (in other words, they are only connected by paths in G_s without type 2 edges)
- $\mathcal{X}_1^\beta \Leftrightarrow$ The variable assignment satisfies (5.1.1)-β, which denotes all those instances of (5.1.1) where a and b are nodes that belong to different instructions (in other words, they are only connected by paths in G_s with one or more type 2 edges)
- $\mathcal{X}_2 \Leftrightarrow$ The variable assignment satisfies (5.1.2)
- $\mathcal{Y}_1 \Leftrightarrow$ The mapping σ satisfies (3.3.2)

- $\mathcal{Y}_2 \Leftrightarrow$ The mapping σ satisfies (3.3.5)

We will successively prove the following four claims:

1. $\mathcal{X}_1^\alpha \wedge \mathcal{X}_2 \Rightarrow \mathcal{Y}_1$
2. $\mathcal{X}_1^\alpha \wedge \mathcal{X}_1^\beta \wedge \mathcal{X}_2 \Rightarrow \mathcal{Y}_2$ (by contradiction)
3. $\mathcal{Y}_1 \Rightarrow \mathcal{X}_1^\alpha \wedge \mathcal{X}_2$
4. $\mathcal{X}_1^\alpha \wedge \neg\mathcal{X}_1^\beta \wedge \mathcal{X}_2 \Rightarrow \neg\mathcal{Y}_2$ (by contradiction)

Equ. 5.1.3 follows from these four claims—the forward implication from the first two and the backward direction from the last two. We commence with the first claim:

. . . .

”$\Rightarrow$”: Let us assume that the assignment of the x variables satisfies both (5.1.1)-α and (5.1.2). It follows from the construction of G_s that the assignment satisfies inequality (5.1.2) with equality—then σ also satisfies (3.3.2) (**Claim 1**).

In order to prove the second claim, we assume that the variable assignment fulfills (5.1.1) and (5.1.2). To show that σ satisfies the precedence constraints (3.3.5), let two arbitrary tuples $(A, s_A) \in \sigma(m)$ and $(B, s_B) \in \sigma(n)$ be given for a DDG edge $(m, n) \in E_D$ such that A and B lie on a control flow path $C \in \mathcal{C}(s(m)) \cap \mathcal{C}(s(n))$. It is sufficient to show that these tuples fulfill (3.3.5). From $x_{(m,A,s_A)} = x_{(n,B,s_B)} = 1$ it follows that there *must not* exist a path from node (m, A, s_A) to (n, B, s_B) in G_s since otherwise this assignment would be excluded by (5.1.1).

We consider the cases $A = B$ and $A \neq B$ separately: If $A = B$, we have to show $(A, s_A + w_{mn}) \preceq (B, s_B)$ (to comply with Equ. (3.3.5)). Suppose for the purpose of contradiction $s_A + w_{mn} > s_B$. Let $t := \min\{s_A, \mathbb{G}_A - w_{mn} + 1\}$, then $1 \leq t \leq \mathbb{G}_A$ follows with Remark 5.1.1-(2) and there exists a path in G_s

- from (m, A, s_A) to (m, A, t) (consisting of type 1 edges, this (possibly empty) subpath exists since $1 \leq t \leq s_A$),
- from there to $(n, B, t+w_{mn}-1)$ (consisting of one type 2 edge; the node $(n, B, t+w_{mn}-1)$ exists since $t + w_{mn} - 1 \leq \mathbb{G}_A$ and $1 \leq t + w_{mn} - 1$ (with $s_A + w_{mn} \geq 2$, which follows from $s_A + w_{mn} > s_B \geq 1$)),
- and from there to (n, B, s_B) (consisting of type 1 edges, this (possibly empty) subpath exists since $t + w_{mn} - 1 = s_A + w_{mn} - 1 \geq s_B$ if $t = s_A$, and $t + w_{mn} - 1 = \mathbb{G}_A \geq s_B$ if $t = \mathbb{G}_A - w_{mn} + 1$).

Because such a path must not exist as pointed out above, we have $s_A + w_{mn} \leq s_B$ and thus $(A, s_A + w_{mn}) \preceq (B, s_B)$. This proves that constraints (3.3.5) are satisfied if $A = B$.

If $A \neq B$, we have to show $A \prec B$. Let us assume the contrary, namely that B is a predecessor of A: Then there exists a path in G_s

- from (m, A, s_A) to $(m, A, 2)$ (consisting of type 1 edges, this (possibly empty) subpath exists since we can assume in this part of the proof $s_A \geq 2$ with Remark 5.1.1-(5)),
- from there to $(n, A, w_{mn} + 1)$ ((consisting of one type 2 edge; the node $(n, A, w_{mn} + 1)$ exists due to Remark 5.1.1-(2)),
- and from there to (n, B, s_B) (consisting of type 1 edges, this subpath exists since $B \in \Theta(n)$ is a predecessor of A)

Again, the existence of this path enforces $x_{(m,A,s_A)} + x_{(n,B,s_B)} \leq 1$, which yields a contradiction. Thus we have shown that σ satisfies the precedence constraints (3.3.5) (**Claim 2**) and completed the first part of the proof.

. . . .

”$\Leftarrow$”: The proof of **Claim 3** is evident: if σ satisfies the assignment constraints (3.3.2), then the assignment of the x variables satisfies (5.1.2) and (5.1.1)-α. For **Claim 4**, let us now assume that the variable assignment satisfies the constraints of the preceding paragraph, but violates an instance of (5.1.1)-β where $a = (m, A, s)$ and $b = (n, B, t)$ are connected by a path P in G_s with one or more type 2 edges (although $x_{(m,A,s)} = x_{(n,B,s)} = 1$). Then it is sufficient to show that σ violates the precedence constraints (3.3.5)—we suppose for the purpose of contradiction that it satisfies them. Let $e_1, \ldots, e_k \in E_s$ be the type 2 edges in P, in the same order as they appear on this path. Let each edge be of the form

$$e_i = \left((n_i, D_i, s_i), (n_{i+1}, D_i, t)\right)$$

with $n_1 = m$, $n_{k+1} = n$, and $B \preceq D_k \preceq D_{k-1} \preceq \ldots \preceq D_1 \preceq A$. We define $D_{k+1} := B$ and say that each type 2 edge e_i *belongs* to the block D_i. Let e_j be the first edge that does not belong to A, but to one of its predecessors (if such an edge does not exist, i.e., if $\forall i : D_i = A$, we set $j := k + 1$). Furthermore, let w_i denote the latency of the DDG edge the type 2 edge e_i is based on. The claim is then proven by means of the following lemma:

Lemma 5.1.8 *For the instructions $n_1, \ldots, n_{k+1}$ holds:*

1. *For all $i \in \{1, \ldots, j\}$, no copy of instruction n_i is scheduled in A at an earlier cycle than $s + \sum_{l<i} w_l$ or in a predecessor of A.*
2. *For all $i \in \{j, \ldots, k + 1\}$, no copy of instruction n_i is scheduled in D_i or in one of its predecessors (if $j \leq k$):*

$$\forall D \preceq D_i : \ \{(E, r) \in \sigma(n_i) \,|E = D\} = \emptyset$$ □

The proof of the lemma is by induction on i. We begin with the **first statement**: The *base case* ($i = 1$) is clear since a copy of n_1 is scheduled at cycle s in A ($x_{(m,A,s)} = 1$). With Prop. 3.3.4 no further copy may be scheduled earlier in A or in a predecessor of A.

To perform the *induction step* $i \to i + 1$, we employ Prop. 5.1.6 with the type 2 edge e_i ($i < j$). We obtain a path $C \in \mathcal{C}(s(n_i)) \cap \mathcal{C}(s(n_{i+1}))$ that passes through A. Copies of both

n_i and n_{i+1} are scheduled on this path in the order imposed by the dependence $(n_i, n_{i+1}) \in E_D$ (since σ is supposed to fulfill (3.3.2) and (3.3.5)). The induction hypothesis states that the copy of n_i on C is not scheduled before cycle $s + \sum_{l<i} w_l$ in A, or in a predecessor of A—consequently, the copy of n_{i+1} on C must be scheduled either in a successor of A or in A at cycle $s + \sum_{l<i+1} w_l$ or later. In addition, with Prop. 3.3.4 no predecessor of A may contain a second copy of n_{i+1}. This concludes the proof of the first statement.

The *base case* of the **second statement** of the lemma ($i = j$) follows from the first statement, which states that no copy of n_j is scheduled in a predecessor of A. D_j is a predecessor of A since there exists a path consisting of type 1 edges from A to D_j (namely the subpath of P between the type 2 edges e_{j-1} and e_j (if $j \geq 2$), or between the node a and e_j (if $j = 1$)).

For the *induction step* $i \to i + 1$ we again employ Prop. 5.1.6 with the type 2 edge e_i. We obtain a path $C \in \mathcal{C}(s(n_i)) \cap \mathcal{C}(s(n_{i+1}))$ that passes through D_i. Again, copies of both n_i and n_{i+1} are scheduled on this path in the order imposed by the dependence $(n_i, n_{i+1}) \in E_D$. Since n_i must not be scheduled in D_i or before on C (the induction hypothesis), the copy of n_{i+1} must be scheduled in a successor of D_i. But then with Prop. 3.3.4 no further copy of n_{i+1} must be scheduled in D_i or in one of its predecessors (like D_{i+1} if $D_i \neq D_{i+1}$)—this was to be shown.

This concludes the proof of the lemma. It is now used to derive a contradiction for each of the two cases $A = B$ and $A \neq B$: If $A = B$, it follows $A = D_1 = D_2 = \ldots = D_{k+1} = B$ and $t \leq s + \sum_{l<k+1}(w_l - 1)$ from the construction of G_s. This contradicts the first statement of the lemma with $i := k + 1$. If $A \neq B$, then the second part with $i := k + 1$ delivers a contradiction with the fact that instruction $n = n_{k+1}$ is scheduled in block $B = D_{k+1}$. ■

The above result that the transitive closure reflects the PCGS-B* problem is neither evident nor easy to achieve in the general case. As a matter of fact, we will later encounter a case where we fail to find a skeleton graph such that the transitive closure constitutes the desired node packing problem. The following lemma reminds why the transitive closure is of such a high importance to us:

Lemma 5.1.9 *The constraint graph G_c is perfect.* □

PROOF It is sufficient to show that G_c is *transitively orientable* (Theorem 4.3.10). A transitive orientation of the edges of G_c (see Def. 4.3.9) is directly given by the corresponding digraph G_s^*. This graph satisfies the transitive property because it is the transitive closure of an acyclic graph G_s. Hence G_c is perfect. ■

The fact that we perform the node packing on a perfect graph directly opens the door to an integral subpolytope of PCGS-B*:

Corollary 5.1.10 (PCGS-B Polytope II) *The following inequalities are equivalent to (5.1.1) and form an integral subpolytope:*

$$\forall \textit{ maximal tournaments } T \subseteq G_s^* : \sum_{a \in N(T)} x_a \leq 1$$

□

PROOF Since the maximal tournaments in G_s^* are exactly the maximal cliques in G_c, the inequalities describe the fractional node-packing polytope of G_c. This polytope is integral because G_c is perfect (with Lemma 5.1.9, Theorem 4.3.8). ■

The statement of this corollary is, taken alone, of little practical value: There can be an exponential number of maximal cliques, and even if this is not the case, the problem remains to find all of them (which is $\mathcal{NP}$-hard in the general case). However, it turns out that by examining the structure of G_s^*, we can easily gain access to all the maximal tournaments: The following two theorems express a direct relationship between these maximal tournaments and the *maximal paths* in the skeleton graph. The first one is directly taken from [CWM94]:

Theorem 5.1.11 *For every maximal tournament $T \subseteq G_s^*$, there exists a unique maximal path $P \subseteq G_s$ such that $N(P) = N(T)$.* □

The proof in [CWM94] relies solely on the fact that G_s^* is the transitive closure of the acyclic graph G_s, but not on the internal structure of G_s. By exploiting this internal structure, however, we can in addition to [CWM94] show that the backward direction holds, too:

Theorem 5.1.12 *For every maximal path $P \subseteq G_s$, there exists a unique maximal tournament $T \subseteq G_s^*$ such that $N(T) = N(P)$.* □

PROOF Each pair of nodes in $N(P)$ is connected by a subpath of P in G_S and thus, due to transitive closure, also by exactly one edge in G_S^*. Hence $T := G_s^*[N(P)]$ (the subgraph of G_s^* induced by $N(P)$) forms a (by definition unique) tournament.

It remains to be shown that the tournament is maximal. Let us assume that a node $u_0 \in N \setminus N(P)$ can be added such that $T' := G_s^*[N(P) \cup \{u_0\}]$ forms a larger tournament. Then, if P is given by the nodes $(u_1, \ldots, u_n)$, for each $k = 1, \ldots, n$ either the edge (u_k, u_0) or (u_0, u_k) exists in the tournament T' by definition.

In particular, either (u_n, u_0) or (u_0, u_n) exists. If (u_n, u_0) exists, then there also exists a path from u_n to u_0 in G_S (because (u_n, u_0) is an edge in the transitive closure of G_S). No inner node on this path may also be on P (otherwise there would be a cycle), hence it can be appended to P in order to form a longer path—in contradiction to the assumption that P is maximal. Hence (u_0, u_n) must exist in T' and, analogically, also the edge (u_1, u_0).

Now let k be the *smallest* index such that (u_0, u_k) is an edge in T'. Because (u_0, u_1) does not exist and (u_0, u_n) does, we know that k exists and that $2 \leq k \leq n$. Moreover, (u_{k-1}, u_0) must exist since (u_0, u_{k-1}) does not exist. Thus we have identified a subgraph in T' as depicted in Fig. 5.3. We denote the path in G_s from u_{k-1} to u_k through u_0 as Q and r as its length.

We will derive a contradiction from the existence of this subgraph to conclude the proof. For this we distinguish four cases:

1. (u_{k-1}, u_k) is a type 1 edge and Q consists only of type 1 edges

2. (u_{k-1}, u_k) is a type 1 edge and Q contains at least one type 2 edge

3. (u_{k-1}, u_k) is a type 2 edge and Q consists only of type 1 edges

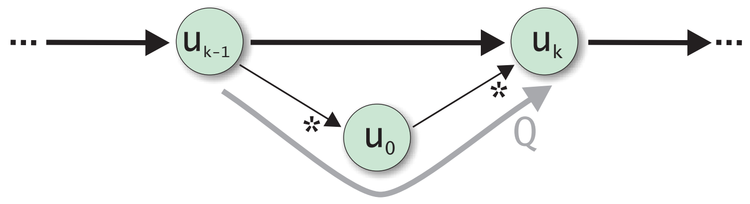


Figure 5.3: Subgraph of T' (Bold edges exist also in G_s).

4. (u_{k-1}, u_k) is a type 2 edge and Q contains at least one type 2 edge

The first case is refuted by Proposition 5.1.5. For the remaining cases, we consider that each of the nodes is a triple (n, B, s) that *belongs* to instruction n, basic block B, and cycle s. Let the sequence $n_Q = (n_1, \ldots, n_r) \in V^r$ denote the instructions and $(s_1, \ldots, s_r) \in \mathbb{N}_+^r$ the cycles of the r nodes on the path Q (in the same order, i.e., s_1 relates to u_{k-1} and s_r to u_k). In the remainder of the proof, we will exploit several properties that are due to the construction of G_S, among them these two facts about two connected nodes (m, A, s) and (n, B, t):

- they are connected by a type 1 edge iff they belong to the same instruction. Then $t = s - 1$ holds if $A = B$.
- they are connected by a type 2 edge iff they belong to different instructions that are connected by a DDG edge. In this case $A = B$ and $t = s + w_{mn} - 1$ hold.

For example, in case 2, n_1 must be equal to n_r because u_{k-1} and u_k are connected by a type 1 edge. However, at least one instruction in n_Q must be different because there is a type 2 edge in Q. Thus the (sub-)sequence of different instructions in n_Q constitutes a nonempty cycle in the DDG and thereby a contradiction. The argument for case 3 goes similarly. For the remaining case 4, we consider the cycle components of the nodes. For (u_{k-1}, u_k) we have:

$$s_r = s_1 + w_{n_1 n_r} - 1 \tag{5.1.4}$$

We deduce a similar relationship between s_1 and s_r using the edges on the path Q. First, we note that u_{k-1} and u_k belong to the same basic block because they are connected by a type 2 edge—then also all the other intermediate nodes on this path must belong to this block as the basic block graph must not contain a cycle. Then we observe that the sequence of all type 2 edges along Q constitutes a DDG path from n_1 to n_r that must have a *total latency less* than $w_{n_1 n_r}$—otherwise the DDG edge (n_1, n_r) would be redundant according to Def. 3.2.7, but we have assumed that the DDG is minimal.

Let w_q denote the mentioned total latency, and let π_1 and π_2 be the total number of type 1 and type 2 edges in Q, respectively. Then we can sum over all edges in Q and estimate:

$$s_r = s_1 + w_q + \pi_1(-1) + \pi_2(-1) \overset{\pi_1+\pi_2 \geq 2}{\leq} s_1 + w_q - 1 \tag{5.1.5}$$

Combining (5.1.4) and (5.1.5) yields:

$$s_r \overset{(5.1.5)}{\leq} s_1 + w_q - 1 < s_1 + w_q \overset{w_q < w_{n_1 n_r}}{\leq} s_1 + w_{n_1 n_r} - 1 \overset{(5.1.4)}{=} s_r$$

This contradiction refutes the last case and concludes the proof. ■

From Theorems 5.1.11 and 5.1.12 it follows that the maximal paths in G_s correspond *exactly* to the maximal tournaments in G_s^*. Thus, to obtain all maximal tournaments and with this an integral polytope, it is sufficient to simply enumerate all the maximal paths. This enumeration can be done recursively in time linear in the number of maximal paths—however, this number itself can still be exponential. At least, it is possible to confine the enumeration to the nodes of individual control flow paths:

Corollary 5.1.13 *For every maximal tournament $T \subseteq G_s^*$, there exists a unique control flow path $C \in \mathcal{C}$ such that $N(T) \subseteq N(C)$.* □

PROOF Proposition 5.1.4 with Theorem 5.1.11. ■

Corollary 5.1.14 (PCGS-B* Polytope III) *The following inequalities form a polytope of PCGS-B*:*

$$\forall \textit{ control flow paths } C \in \mathcal{C}, \forall \textit{ maximal paths } P \subseteq G_s[N(C)] : \sum_{a \in N(P)} x_a \leq 1 \tag{5.1.6}$$

$$\forall n \in V, \forall \textit{ paths } C \in \mathcal{C}(s(n)) : \sum_{a \in N(C) \cap N(n)} x_a \geq 1 \tag{5.1.7}$$

The subpolytope described by inequalities (5.1.6) is integral. □

PROOF The preceding corollary with Theorem 5.1.11. ■

Before we tackle the exponential complexity of this formulation, we examine which impact the additional inequalities (5.1.7) have on the integrality of the whole polytope. For this purpose the following observation is helpful: all the left-hand sides of instances of (5.1.7) occur also as left-hand sides of (5.1.6), namely as exactly those left-hand sides related to paths $P \subseteq G_s[N(C)]$ *without* type 2 edges. This is expressed by the following lemma, where $\tau_2(P)$ denotes the number of type 2 edges in P:

Lemma 5.1.15 *For all control flow paths $C \in \mathcal{C}$, all maximal paths $P \subseteq G_s[N(C)]$ holds:*

$$\tau_2(P) = 0 \Leftrightarrow \exists n \in V : N(P) = N(C) \cap N(n)$$ □

PROOF ”⇒”: If $\tau_2(P) = 0$, then P consists only of type 1 edges and there must be an instruction n to which all nodes on P belong, i.e., such that $N(P) \subseteq N(C) \cap N(n)$. Note that the sub-graph $\hat{P} = G_s[N(C) \cap N(n)]$ constitutes a maximal path in G_s. Hence $P \subseteq \hat{P}$, and since both paths are maximal, we have $P = \hat{P}$ and with this $N(P) = N(\hat{P}) = N(C) \cap N(n)$.

”⇐”: Since all nodes of P belong to one instruction, it cannot contain any type 2 edges. ■

The lemma allows us to conclude that the polytope remains the same if we omit the constraints (5.1.7) and instead change the “$\leq$” relation symbol in (5.1.6) to “$=$” for those instances where $\tau_2(P) = 0$. In doing so, we preserve the integrality of the subpolytope defined by (5.1.6), as guaranteed by Theorem 4.3.11. Thus we have:

Theorem 5.1.16 (PCGS-B* Polytope IV) *The following constraints form an integral polytope of PCGS-B*:*

$$\begin{array}{r} \forall \textit{ control flow paths } C \in \mathcal{C}, \\ \forall \textit{ maximal paths } P \subseteq G_s[N(C)] \end{array} : \begin{cases} \sum_{a \in N(P)} x_a \leq 1 & \textit{if } \tau_2(P) \geq 1 \\ \sum_{a \in N(P)} x_a = 1 & \textit{if } \tau_2(P) = 0 \end{cases} \tag{5.1.8}$$

□

This formulation’s conciseness should not belie the fact that its complexity is twofoldly exponential: firstly in the number of control flow paths and secondly in the number of maximal paths in $G_s[N(C)]$.

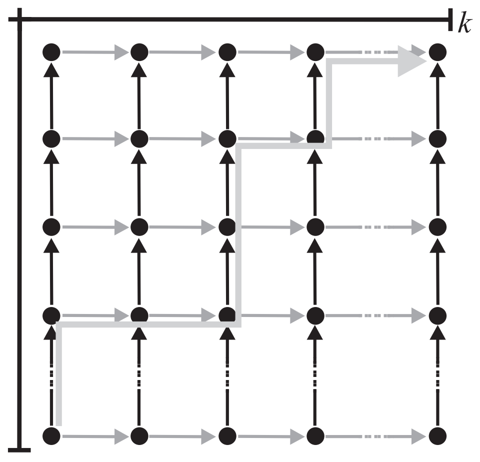


Figure 5.4: Estimating the number of maximal paths in a skeleton digraph.

Figure 5.4 provides a more detailed analysis of the latter number: it shows the nodes of an (artificially constructed) skeleton digraph for a single basic block with l cycles and a sequence of k successively dependent instructions (all with latency 1). This graph has a checkerboard structure, where the type 1 edges go vertically upwards and the type 2 edges go horizontally rightwards. Each maximal path goes from the left lowermost to the right uppermost node in the graph. The number of these paths is exactly[1]

$$\binom{l+k-2}{k-1} \overset{\text{With [CLR01]}}{\geq} \left(\frac{l+k-2}{k-1}\right)^{k-1} = \left(1 + \frac{l-1}{k-1}\right)^{k-1}$$

[1]Each maximal path is uniquely defined by the numbers of its type 2 edges at each of the l cycles (between 0 and $k-1$ per cycle, $k-1$ altogether). Hence the number of possible paths corresponds to the number of possibilities to distribute $k-1$ indistinguishable balls ($\approx$ type 2 edges) among l urns ($\approx$ cycles) of unlimited size, which is $\binom{l+k-2}{k-1}$.

and hence exponential in k if $l = \Omega(k)$, which is a realistic possibility. However, if we could bound the number of type 2 edges allowed in the maximal paths by a constant, no exponential growth would occur. This is exactly what the next step does.

5.1.2 Reducing the Complexity of the Integral Subpolytope

In this section, we will—one by one—eliminate the two sources of exponential complexity in our derived polytope of PCGS-B*, while keeping it integral. As indicated before, we can confine the scope of maximal paths examined during the constraint generation: any path with more than one type 2 edge is superfluous, as expressed by the following theorem from [CWM94]. Due to the importance of this theorem we provide a version of the proof from [CWM94] that is adapted to our own concepts and notions in Appendix B.2.1.

Theorem 5.1.17 *Any of the constraints (5.1.8) for a maximal path with $\tau_2(P) \geq 2$ is a linear combination of those constraints with $\tau_2(P) < 2$.* □

As a result of this theorem, we can remove an exponential number of redundant constraints from the formulation without altering the polytope. The resulting formulation is given by the following corollary, where constraints (5.1.10) and (5.1.11) correspond to (5.1.8) for maximal paths with one and zero type 2 edges, respectively.

We provide a more detailed derivation of the inequalities (5.1.10) since during the transition from (5.1.8) to (5.1.10) the deficiency from Remark 5.1.1-(5) is fixed. Let $N_t \subseteq N$ be the set of those nodes with cycle component equal to t, then we simply replace all variables x_a such that $a \in N_1$ by zero to exclude the possibility to schedule instructions there. This replacement cures the deficiency and preserves the integrality of the polytope with Corollary 4.3.12. For reasons of symmetry, we also replace all nodes of the last cycle, $N_{\mathbb{G}_A}$, by zero; this just serves to simplify the resulting formulas.

The replacement is allowed for in the sums of the following inequalities (5.1.9), which correspond otherwise to the constraints (5.1.8) for all those paths with exactly one type 2 edge. They are instantiated for all type 2 edges in G_s, however, their creation is described independently of G_s here—from now on we will cease using the node packing representation of the problem. The left and the right double sums add up the variables of the nodes on the path before and after the type 2 edge $((m, A, t), (n, A, t + w_{mn} - 1))$, respectively:

$$\sum_{\substack{a \in N(C) \cap N(m) \setminus \left(N_1 \cup N_{\mathbb{G}_A}\right) \\ a \succeq (m,A,t)}} x_a + \sum_{\substack{a \in N(C) \cap N(n) \setminus \left(N_1 \cup N_{\mathbb{G}_A}\right) \\ a \preceq (n,A,t+w_{mn}-1)}} x_a \leq 1 \tag{5.1.9}$$

$$\forall (m,n) \in E_D,\ \forall C \in \mathcal{C}(s(n)) \cap \mathcal{C}(s(m)),\ \forall A \in \mathcal{B}(C),\ \forall t \in \{1, \ldots, \mathbb{G}_A - w_{mn} + 1\}$$

The used order of the nodes corresponds to that of the scheduling positions (3.3.1):

$$(m, A, s_A) \prec (n, B, s_B) \Leftrightarrow \begin{cases} A \prec B \text{ in } G_B & \text{if } A \neq B \\ s_A < s_B & \text{if } A = B \end{cases}$$

Due to the exclusion of x variables of the first and last cycle from the sums we can shrink the range of t in Equ. (5.1.9) to $\{2, \ldots, \mathbb{G}_A - w_{mn}\}$ without changing the generated constraints. In the following version (5.1.10), we have furthermore

- removed all nodes in $N_1 \cup N_{\mathbb{G}_A}$ from N and decremented the cycle component of the remaining nodes (and the range of t) by one, so that the former nodes of the i-th cycle become the new nodes of the $(i-1)$-th cycle, and

- incremented $\mathbb{G}_A$ by two to compensate for the removed nodes of two cycles.

Corollary 5.1.18 (PCGS-B Polytope $\mathbb{V}$) *The following inequalities form an integral polytope of PCGS-B:*

$$\sum_{\substack{a \in N(C) \cap N(m) \\ a \succeq (m,A,t)}} x_a + \sum_{\substack{a \in N(C) \cap N(n) \\ a \preceq (n,A,t+w_{mn}-1)}} x_a \leq 1 \tag{5.1.10}$$

$$\forall (m,n) \in E_D,\ \forall C \in \mathcal{C}(s(n)) \cap \mathcal{C}(s(m)),\ \forall A \in \mathcal{B}(C),\ \forall t \in \{1, \ldots, \mathbb{G}_A - w_{mn} + 1\}$$

$$\forall n \in V,\ \forall C \in \mathcal{C}(s(n)) : \sum_{a \in N(C) \cap N(n)} x_a = 1 \tag{5.1.11}$$

□

This formulation is a big leap forwards, but it still grows exponentially with the number of control flow paths. To cope with this effect, we exploit a different form of redundancy in the formulation that is described by the following lemma. It holds for all solution vectors inside the polytope and says informally that, for any instruction n and any candidate block A, the sum of all the x variables of n on a path from an entry block to A has the same value for all such paths.

Lemma 5.1.19 *For every real-valued solution of (5.1.11), every instruction $n \in V$ and every block $A \in \Theta(n)$ there exists a unique real number $\alpha_n^A \in [0,1]$ such that:*

$$\forall C \in \mathcal{C}(A) \cap \mathcal{C}(s(n)) : \sum_{\substack{a=(n,B,s) \in N(C) \cap N(n) \\ B \prec A}} x_a = \alpha_n^A \tag{5.1.12}$$

□

PROOF Let there be any two distinct paths $C, D \in \mathcal{C}(A) \cap \mathcal{C}(s(n))$ given. We show that for both paths the left-hand side of the above equation must have the same value under the constraints (5.1.11). For this purpose we construct a third path $E \in \mathcal{C}(A) \cap \mathcal{C}(s(n))$ as a concatenation of two subpaths: the subpath of C containing only predecessors of A plus the subpath of D containing only A and its successors (see Fig. 5.5).

This path E traverses $s(n)$ since either the first or the second subpath must traverse $s(n)$: if $s(n) \prec A$, then the first subpath (of $C \in \mathcal{C}(s(n))$) passes through $s(n)$; if $s(n) \succeq A$, then the

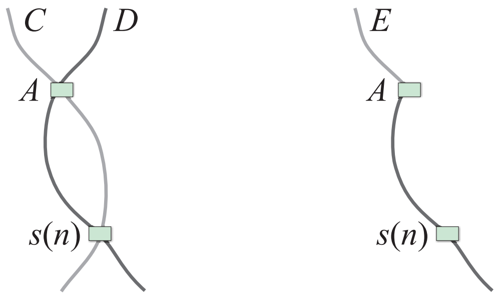


Figure 5.5: Construction of E (right-hand side) from the paths C and D (left-hand side).

second subpath (of $D \in \mathcal{C}(s(n))$) passes through $s(n)$. Hence instances of (5.1.11) are created for both D and E, respectively, and can be written as:

$$\sum_{\substack{a=(n,B,s)\in N(D)\cap N(n)\\ B\prec A}} x_a + \sum_{\substack{a=(n,B,s)\in N(D)\cap N(n)\\ B\succeq A}} x_a = 1$$

$$\sum_{\substack{a=(n,B,s)\in N(C)\cap N(n)\\ B\prec A}} x_a + \sum_{\substack{a=(n,B,s)\in N(D)\cap N(n)\\ B\succeq A}} x_a = 1$$

Combining these two equations gives

$$\sum_{\substack{a=(n,B,s)\in N(C)\cap N(n)\\ B\prec A}} x_a = \sum_{\substack{a=(n,B,s)\in N(D)\cap N(n)\\ B\prec A}} x_a$$

which shows that α_n^A exists and is unique. ■

From now on we assume that α_n^A is always defined as above in the context of a solution of (5.1.11). This is also the case in the following corollary:

Corollary 5.1.20 *For every real-valued solution of (5.1.11), every instruction $n \in V$, every block $A \in \Theta(n)$, and every direct successor $B \in \Theta(n)$ of A holds:*

$$\forall C \in \mathcal{C}(A) \cap \mathcal{C}(B) \cap \mathcal{C}(s(n)) : \sum_{\substack{a=(n,D,s)\in N(C)\cap N(n)\\ D\succ A}} x_a = 1 - \alpha_n^B \tag{5.1.13}$$

□

PROOF Very similar to the proof of the preceding lemma. ■

We observe that the left-hand sides of (5.1.12) and (5.1.13) are also contained in the left-hand sides of Equ. (5.1.10) and (5.1.11) of the current ILP formulation. They represent there—for each instruction and each basic block—an exponential number of different expressions that must have the same value in any real-valued solution. To exploit this equivalence in order to simplify the formulation, we introduce a new binary variable $a_n^{\uparrow A}$ for α_n^A and replace in the formulation of Corollary 5.1.18 all occurrences of the left-hand sides of (5.1.12) and (5.1.13) by $a_n^{\uparrow A}$ and $1-a_m^{\uparrow B}$, respectively. The resulting *polynomial sized* formulation is shown in the following theorem, where the constraints (5.1.15) are the adapted constraints (5.1.10); the so-called a-x *constraints* (5.1.16) together with (5.1.17) form the new assignment constraints (resulting from (5.1.11)). Beginning with this theorem, we use the notation x_n^{At} instead of $x_{(n,A,t)}$ and this abbreviation:

$$G_{mn}(A) := \{1, \ldots, \mathbb{G}_A - w_{mn} + 1\} \tag{5.1.14}$$

Theorem 5.1.21 (PCGS-B Polytope $\underline{\mathrm{VI}}$) *The following constraints form a polytope of PCGS-B:*

$$a_n^{\uparrow A} + \sum_{\substack{t_n \in G(A) \\ t_n \le t + w_{mn} - 1}} x_n^{At_n} + \sum_{\substack{t_m \in G(A) \\ t_m \ge t}} x_m^{At_m} + (1 - a_m^{\uparrow B}) \le 1 \tag{5.1.15}$$

$$\forall (m,n) \in E_D,\ \forall A \in \Theta(m) \cap \Theta(n),\ \forall t \in G_{mn}(A)$$

$$\forall n \in V,\ \forall A, B \in \Theta(n) \text{ such that } (A,B) \in E_B :\ a_n^{\uparrow B} = a_n^{\uparrow A} + \sum_{t \in G(A)} x_n^{At} \tag{5.1.16}$$

$$\forall n \in V :\ a_n^{\uparrow \Omega} = 1 \tag{5.1.17}$$

In (5.1.15), B denotes a direct successor block of A that is element of $\Theta(m)$. If no such block exists, the whole term $(1 - a_m^{\uparrow B})$ is omitted. For all entry blocks $A \in \mathcal{B}_{entry}$, the variable $a_n^{\uparrow A}$ is replaced by zero in (5.1.15) and (5.1.16). □

PROOF The proof can be done by showing that the schedules described by these constraints are exactly the same as those from Corollary 5.1.18, so it is sufficient to show that for each vector in polytope $\underline{\mathrm{V}}$ there exists a corresponding vector in polytope $\underline{\mathrm{VI}}$ that represents the same schedule *and vice versa*. This is expressed by the last of the following three claims, which are stated with respect to two vectors $\overrightarrow{x} = (x_a)_{a \in N} \in [0,1]^+$ and $\overrightarrow{x_*} = \left((x_n^{At})_{\forall A \forall t \forall n}, (a_n^{\uparrow A})_{\forall A \forall n} \right) \in [0,1]^+$ that represent the same schedule, i.e., $\forall n \forall A \forall t :\ x_n^{At} = x_{(n,A,t)}$:[2]

1. $\overrightarrow{x_*}$ satisfies (5.1.16) if and only if $\forall n \forall A : \alpha_n^A = a_n^{\uparrow A}$.
2. $\overrightarrow{x_*}$ satisfies (5.1.16) and (5.1.17) if and only if $\overrightarrow{x}$ satisfies (5.1.11) and $\forall n \forall A : \alpha_n^A = a_n^{\uparrow A}$.
3. $\overrightarrow{x_*}$ satisfies (5.1.16), (5.1.17), and (5.1.15) if and only if $\overrightarrow{x}$ satisfies (5.1.11), (5.1.10), and $\forall n \forall A : \alpha_n^A = a_n^{\uparrow A}$.

We successively prove these claims. The backward direction of the **first claim** is straightforward: we have $\forall n \forall A : \alpha_n^A = a_n^{\uparrow A}$ and hence for each instruction n, each block $B \in \Theta(n)$ and each direct predecessor $A \in \Theta(n)$:

$$a_n^{\uparrow B} = \alpha_n^B \overset{\text{By definition of } \alpha_n^B}{=} \alpha_n^A + \sum_{t \in G(A_i)} x_n^{At} = a_n^{\uparrow A} + \sum_{t \in G(A_i)} x_n^{At}$$

[2]It would be sufficient for the proof to restrict $\overrightarrow{x}$ and $\overrightarrow{x_*}$ to integer vectors from $\{0,1\}^+$. However, we show a stronger result for real-valued vectors which can be utilized in a later proof.

The proof of the forward implication is by induction on the depth of the blocks in the basic block graph: For the base case, let B be an entry block, then we have $\forall n : \alpha_n^A = a_n^{\uparrow A} = 0$. To perform the induction step, let an $n \in V$ and a block $B \in \Theta(n)$ with direct predecessors $A_1, \dots, A_k$ be given: For all $i \in \{1, \dots, k\}$ we have $a_n^{\uparrow B} = a_n^{\uparrow A_i} + \sum_{t \in G(A_i)} x_n^{A_i t}$ and, by the induction hypothesis, $\alpha_n^{A_i} = a_n^{\uparrow A_i}$. It follows for any path $C \in \mathcal{C}(B) \cap \mathcal{C}(s(n))$, which must traverse a predecessor A_j of B:

$$\sum_{\substack{a=(n,D,s)\in N(C)\cap N(n)\\ D\prec B}} x_a \overset{\left(x_n^{At}=x_{(n,A,t)}\right)}{=} \sum_{\substack{a=(n,D,s)\in N(C)\cap N(n)\\ D\prec A_j}} x_a + \sum_{t\in G(A_j)} x_n^{A_j t} \overset{\left(\alpha_n^{A_j}=a_n^{\uparrow A_j}\wedge(5.1.16)\right)}{=} a_n^{\uparrow B}$$

This equality implies that α_n^B is well defined and equal to $a_n^{\uparrow B}$, which concludes the proof of the first claim. With its help we can now directly prove the **second claim** (using the notation $\overrightarrow{x_*} \models$ (5.1.16) for "$\overrightarrow{x_*}$ satisfies (5.1.16)" etc.) :

$$\begin{aligned}
& \overrightarrow{x_*} \models (5.1.16) \wedge \forall n \in V : \; a_n^{\uparrow\Omega} = 1 \\
\overset{(1)}{\Leftrightarrow} \;& \forall A \forall n : \alpha_n^A = a_n^{\uparrow A} \wedge \forall n \in V : \; \alpha_n^\Omega = 1 \\
\Leftrightarrow \;& \forall A \forall n : \alpha_n^A = a_n^{\uparrow A} \wedge \forall n \in V, \forall C \in \mathcal{C}(\Omega) \cap \mathcal{C}(s(n)) : \sum_{\substack{a=(n,B,s)\in N(C)\cap N(n)\\ B\prec\Omega}} x_a = 1 \\
\Leftrightarrow \;& \forall A \forall n : \alpha_n^A = a_n^{\uparrow A} \wedge \overrightarrow{x} \models (5.1.11)
\end{aligned}$$

In the last step we have used the fact that Ω is the single empty exit block, which must be traversed by all paths. The second claim (2) can now be used to prove the **third claim**, and with this the theorem:

$$\begin{aligned}
& \overrightarrow{x_*} \models (5.1.16), (5.1.17) \wedge \forall (m,n) \in E_D, \, \forall A \in \Theta(m) \cap \Theta(n), \, \forall t \in G_{mn}(A) : \\
& a_n^{\uparrow A} + \sum_{\substack{t_n \in G(A)\\ t_n \le t + w_{mn} - 1}} x_n^{A t_n} + \sum_{\substack{t_m \in G(A)\\ t_m \ge t}} x_m^{A t_m} + (1 - a_m^{\uparrow B}) \le 1 \\
\overset{(2)}{\Leftrightarrow} \;& \forall A \forall n : \alpha_n^A = a_n^{\uparrow A} \wedge \overrightarrow{x} \models (5.1.11) \wedge \forall (m,n) \in E_D, \, \forall A \in \Theta(m) \cap \Theta(n), \, \forall t \in G_{mn}(A) : \\
& \alpha_n^A + \sum_{\substack{t_n \in G(A)\\ t_n \le t + w_{mn} - 1}} x_n^{A t_n} + \sum_{\substack{t_m \in G(A)\\ t_m \ge t}} x_m^{A t_m} + (1 - \alpha_n^B) \le 1 \\
\Leftrightarrow \;& \forall A \forall n : \alpha_n^A = a_n^{\uparrow A} \wedge \overrightarrow{x} \models (5.1.11) \wedge \forall (m,n) \in E_D, \, \forall C \in \mathcal{C}(s(n)) \cap \mathcal{C}(s(m)), \\
& \forall A \in \mathcal{B}(C), \, \forall t \in G_{mn}(A) : \\
& \sum_{\substack{a \in N(C)\cap N(m)\\ a \succeq (m,A,t)}} x_a + \sum_{\substack{a \in N(C)\cap N(n)\\ a \preceq (n,A,t+w_{mn}-1)}} x_a \le 1 \\
\Leftrightarrow \;& \forall A \forall n : \alpha_n^A = a_n^{\uparrow A} \wedge \overrightarrow{x} \models (5.1.11), (5.1.10)
\end{aligned}$$

In the second last step, we have used Corollary 5.1.20 as well as the equality $\Theta(m) \cap \Theta(n) = \{A \in \mathcal{B}(C) \,|\, C \in \mathcal{C}(s(n)) \cap \mathcal{C}(s(m))\}$ from Remark 5.1.1-(1). ■

With this new polytope $\mathbb{VI}$ we have found a polynomial sized description of PCGS-B. We observe that, due to the introduction of the new a variables, the new polytope is of higher dimension than the original polytope $\mathbb{V}$. More precisely, we can imagine that the original polytope is contained in a subspace of this higher-dimensional space (if the x_n^{At} axes are equivalent to the $x_{(n,A,t)}$ axes), namely in that subspace where all the a variables are zero. Then a notable consequence of Theorem 5.1.21 is that the original polytope $\mathbb{V}$ matches the *projection* of the new polytope $\mathbb{VI}$ onto this subspace (for the definition refer to Sec. 4.3).

Thus we have simplified the polytope (with regard to the number of facets or constraints) by “lifting” it to a higher-dimensional space—the projection of the resulting polynomial sized polytope onto the original space then produces the original, exponential sized polytope. Interestingly, we will encounter the same phenomenon later again with a different formulation (and in the opposite direction).

A serious consequence of the lifting is, however, that the integrality result is lost: the new polytope is no longer a node-packing polytope, which removes the fundament on which the integrality proof was based. But the above theorem has shown a deep similarity between the original integral polytope and the new one: there is a one-to-one mapping not only between integral points inside these polytopes, but even between real-valued points. This kind of isomorphism suggests that the lifting has possibly preserved the integrality of the original polytope. Indeed, by applying the following theorem we can exploit the similarity to regain the integrality result:

Theorem 5.1.22 *Let $P \subset \mathbb{R}^m$ and $Q \subset \mathbb{R}^n$ be two polytopes where P is integral, and let $f : P \to Q$ be an affine bijection with integral coefficients:*

$$x \mapsto Ax + b, \quad A \in \mathbb{Z}^{n\times m},\, b \in \mathbb{Z}^n$$

Then Q is integral. □

PROOF We first show that for all $x \in P$ holds: if $f(x)$ is an extreme point in Q, then x is an extreme point in P. We assume the opposite: let $f(x)$ be an extreme point and x not, i.e., let there exist $w, y \in P, w \neq y$ and a $c \in]0,1[$ such that $cw + (1-c)y = x$. Then since f is affine $f(x) = f(cw + (1-c)y) = cf(w) + (1-c)f(y)$ holds, and from the injectivity of f it follows $f(w) \neq f(y)$. This contradicts the assumption that $f(x)$ is an extreme point.

Now let an extreme point $y \in Q$ be given. Since f is surjective there exists an $x \in P$ such that $f(x) = y$ and x is an extreme point. Then x is integral since P is integral, and $f(x) = y$ must be integral as well since the affine map has only integral coefficients. Hence Q is integral. ■

Corollary 5.1.23 *The polytope $\mathbb{VI}$ of PCGS-B from Theorem 5.1.21 is integral.* □

PROOF We apply Theorem 5.1.22 with P as the polytope $\mathbb{V}$ from Corollary 5.1.18 and Q as the polytope $\mathbb{VI}$ from Theorem 5.1.21. The proof of Theorem 5.1.21 already implies a function $f : P \to Q$ that maps a vector $\overrightarrow{x} = (x_a)_{a\in N} \in [0,1]^+$ on a vector $\overrightarrow{x_*} = \left(\left(x_n^{At}\right)_{\forall A\forall t\forall n}, \left(a_n^{\uparrow A}\right)_{\forall A\forall n}\right) \in [0,1]^+$such that $\forall n \forall A \forall t$:

- x_n^{At} is assigned $x_{(n,A,t)}$ and

- $a_n^{\uparrow A}$ is assigned $\sum_{\substack{a=(n,B,s)\in N(C)\cap N(n)\\ B\prec A}} x_a$ for an (arbitrarily chosen) path $C \in \mathcal{C}(A) \cap \mathcal{C}(s(n))$ (i.e., the value α_n^A)

Clearly f is linear and has integral coefficients. In addition, Claim 3 inside the proof of Theorem 5.1.21 shows that the function is total and surjective. Finally, it is also injective because $\forall n \forall A \forall t$: $x_{(m,A,s)} \neq x_{(n,B,t)} \Rightarrow x_m^{As} \neq x_n^{Bt}$ and the values of α_n^A are unique (Lemma 5.1.19). ■

With this proof we have arrived at our goal of obtaining an efficient polytope of PCGS-B: the polynomial sized ILP formulation from Theorem 5.1.21 is integral—a result that is also confirmed empirically: We have generated ILPs of this subproblem for each of the input programs from Chapter 7 and solved iteratively the LP-relaxation with different, randomized objective functions (a hundred times for each input). The returned extreme points of the PCGS-B polytope were always integral.

Before we go about integrating the block length constraints we will examine the functioning of this ILP formulation and provide a more meaningful interpretation of the a variables. The semantics of the x variables is clear:

$x_n^{At} = 1 \quad \Leftrightarrow \quad$ A copy of instruction n is scheduled at cycle t in block A.

Lemma 5.1.19 implies for all feasible solutions of the ILP the following semantics:

$a_n^{\uparrow A} = 1 \quad \Leftrightarrow \quad$ A copy of instruction n is scheduled on each program path through $s(n)$ before A.

The constraint $a_n^{\uparrow\Omega} = 1$ (5.1.17) expresses that every path through the source block of an instruction n must encounter a scheduled (compensation) copy of this instruction. Fig. 5.6 illustrates this[3] using an example: Along each path starting from Ω upwards through the source block, the a variables are equal to one up to the block with the scheduled copy—from there on, they are and remain zero-valued, and consequently no further copy of the instruction can be scheduled along the path according to Equ. 5.1.16. Along the path, the values of the a variables are—also for real-valued solutions—monotonically decreasing:

Proposition 5.1.24 *Let an instruction $n \in V$ and a program path through $s(n)$ be given. Let $A_1, \ldots, A_k$ be the sequence of blocks encountered when moving along this path in the opposite direction, starting from $A_1 = \Omega$. Along this sequence, the value of $\sum_{t\in G(A_i)} x_n^{A_i t} + (1 - a_m^{\uparrow A_{i-1}})$ is monotonically increasing and the value of $a_n^{\uparrow A_i}$ is monotonically decreasing (in any real-valued feasible solution). The sum of these terms is always equal to one.* □

PROOF Follows inductively from the assignment constraints (5.1.17) and (5.1.16). The first and the second term are equal to the sum of the x_n variables[4] of the blocks $A_1, \ldots, A_i$ and $A_{i+1}, \ldots, A_k$, respectively. ■

[3]In Fig. 5.6 (b) and in all following illustrations of schedules, those cycles where instructions are scheduled are shown in a darker shade. Thus the shaded parts represent the schedule lengths of the blocks.

[4]While the notion "x variables" refers to all generated x_n^{At} variables, $\{x_n^{At} \mid \forall n \in V, \forall A \in \Theta(n), \forall t \in G(A)\}$, "$x_n$ variables" is used to refer to those variables only that belong to instruction n: $\{x_n^{At} \mid \forall A \in \Theta(n), \forall t \in G(A)\}$.

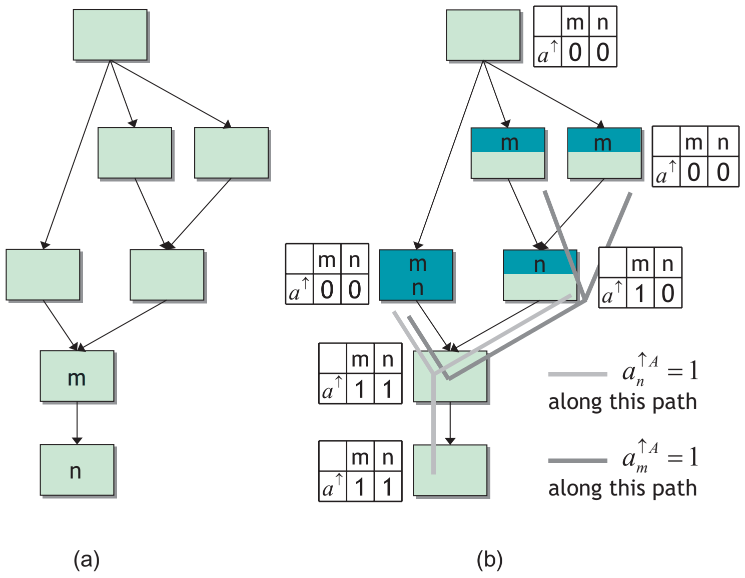


Figure 5.6: Semantics of the a variables: A global scheduling problem with two data dependent instructions ($(m, n) \in E_D$) in their respective source blocks (a) and a feasible schedule (b).

In an earlier version of the model, we have generated the following distinct *local* and *global* precedence constraints [Win02, Win04]:

$$\sum_{\substack{t_n \in G(A) \\ t_n \le t + w_{mn} - 1}} x_n^{At_n} + \sum_{\substack{t_m \in G(A) \\ t_m \ge t}} x_m^{At_m} \le 1 \qquad \begin{array}{l} \forall (m,n) \in E_D, \\ \quad \forall A \in \Theta(m) \cap \Theta(n),\ \forall t \in G_{mn}(A) \end{array} \tag{5.1.18}$$

$$a_n^{\uparrow A} \le a_m^{\uparrow A} \qquad \forall (m,n) \in E_D,\ \forall A \in \Theta(m) \cap \Theta(n) \tag{5.1.19}$$

This is not necessary in the current version since all instances of the precedence constraints (5.1.15) for $t := \mathbb{G}_A - w_{mn} + 1$ subsume inequalities of the form:

$$\underbrace{a_n^{\uparrow A} + \sum_{t_n \in G(A)} x_n^{At_n}}_{= a_n^{\uparrow B}} + (1 - a_m^{\uparrow B}) \le 1 \Leftrightarrow a_n^{\uparrow B} \le a_m^{\uparrow B} \tag{5.1.20}$$

These are exactly the former global precedence constraints (5.1.19)—they are special cases of the new, generalized precedence constraints (5.1.15). The local constraints (5.1.18) are trivially subsumed. The set of integer feasible solutions is unchanged if the separated constraints (5.1.18) and (5.1.19) are used in place of (5.1.15). But the resulting polytope is “looser” and no longer integral: When its extreme points are scanned via randomized objective functions as described above, a part of the obtained points is fractional-valued (less than a half for most inputs). Nevertheless, it can be appropriate to employ the separated variant if it is useful to distinguish between local and global preservation of data dependences (see Sec. 6.3).

Now we consider possibilities to integrate the block length variables and constraints into the PCGS-B polytope, preferably while not destroying its integrality. The classic way represents the block lengths directly by integer variables [Käs00b, GE93]. For this, we can introduce for each basic block A a new integer (but not binary) variable T_A that is greater than or equal to its length in the schedule:

$$\sum_{t \in G(A)} t \cdot x_n^{At} \leq T_A \qquad \forall n \in V,\ \forall A \in \Theta(n) \tag{5.1.21}$$

These constraints work as follows: If block A has length $\tilde{t}$ and n is an instruction scheduled in this block at cycle $\tilde{t}$, then the left-hand side of the inequality instantiated for this instruction and this block evaluates to $\tilde{t}$ since only $x_n^{A\tilde{t}}$ is equal to one in the sum—$\tilde{t} \cdot x_n^{A\tilde{t}} = \tilde{t}$ is the only non-zero addend. With the execution frequency of block A given as f_A, the objective function can be written as:

$$\min_{A \in \mathcal{B}} \sum f_A \cdot T_A \tag{5.1.22}$$

This objective function is such that in each optimal solution, for each block A at least one instance of inequality (5.1.21) is tight so that T_A is equal to the actual block length. The advantage of this approach is that it needs only $\mathcal{O}(|\mathcal{B}|)$ variables and $\mathcal{O}(|\mathcal{B}| \cdot |V|)$ constraints. But it has the significant drawback that its efficiency is unclear and difficult to analyze; at least, the experiments show that it is not integral.

An alternative formulation introduces a new set of binary variables for each block to represent its length. Such a variable B_t^A is equal to one if and only if basic block A has length t in the schedule. This variable can be interpreted as the variable $x_{l_A}^{At}$ of a new imaginary "last" instruction l_A inside block A that is dependent on *all* other instructions that are scheduled in A (with latency zero). Accordingly, we can use instances of the local precedence constraints (5.1.18) to link the B_t^A variables to the model. The objective function remains linear:

$$\min \sum_{A \in \mathcal{B}} f_A \cdot \left(\sum_{t \in G(A)} t \cdot B_t^A \right) \tag{5.1.23}$$

If block A has length $\tilde{t}$, then in any optimal solution the term in the parentheses evaluates to $\tilde{t}$ (because then $\tilde{t} \cdot B_{\tilde{t}}^A = \tilde{t}$ is the only non-zero addend). This formulation is, with $\mathcal{O}(\mathbb{G})$ variables and $\mathcal{O}(\mathbb{G} \cdot |V|)$ constraints, larger than the previous one, namely by a factor that corresponds approximately to the average maximum block length. But the experiments indicate that it is tighter; however, fractional-valued solutions still occur in the relaxation—hence the resulting polytope is not integral.

To obtain deeper insights into its integrality properties, we tried to transfer the proofs developed in Sec. 5.1; however, we encountered two obstacles: Firstly, it seems impossible to extend the skeleton digraph in such a way that its transitive closure yields a constraint graph that models the logical characteristics of the B variables (= the x variables of l_A) properly. The problem is that the pseudo instruction l_A of a block A should be dependent on other instructions only if they are scheduled in A, i.e., these dependences may only be *local*[5]. However, by taking the transitive closure, all dependences are automatically translated to all successor blocks—in other words, they are made global. This unwanted consequence must not be an obstacle in principle to obtaining a correct and perfect constraint graph, it could also merely be a limitation of our proof method.

Secondly, even if the B variables are correctly integrated into the constraint graph, it is unclear if and to which extent Theorem 5.1.17 can be adapted to local dependences. In fact, examples of constraints (5.1.8) with $\tau_2(P) \geq 2$ involving local dependences can be found that are apparently *not* redundant. Thus an exponential worst case number of these constraints could remain.

As all attempts to cope with this complexity do not succeed, the question arises whether a fundamental barrier has been reached here. In fact—and perhaps surprisingly—we can prove that, if $\mathcal{P} \neq \mathcal{NP}$, any attempt to integrate the block length constraints into the polytope while keeping it efficient is bound to fail:

Theorem 5.1.25 *Precedence-constrained global instruction scheduling (PCGS) is $\mathcal{NP}$-complete.* □

PROOF It is clear that PCGS is in $\mathcal{NP}$: It is surely possible to check in polynomial time whether a given schedule satisfies the assignment, precedence, and block length constraints. To show the $\mathcal{NP}$-hardness, we reduce the Max2SAT problem to PCGS:

Definition 5.1.26 (Max2SAT Problem) Let a boolean formula in conjunctive normal form be given where each clause consists of at most two literals. Given an integer $k \leq n$, is there a truth assignment that satisfies at least k clauses? □

Max2SAT is $\mathcal{NP}$-complete[6] [GJ79]. Let an instance of this problem be given by a formula $C = C_1 \wedge \ldots \wedge C_n$ and an integer $k \leq n$. We assume that each clause C_i has exactly two literals L_{i1} and L_{i2} (otherwise we can duplicate single literals). Our goal is to construct an instance of PCGS, called $PCGS(C)$, that has a minimal schedule length less than a certain value if and only if there exists a truth assignment for C that satisfies at least k clauses.

The choice to assign either true or false to a variable must be incorporated into $PCGS(C)$. For this purpose we employ for each variable a PCGS subproblem that represents its truth assignment. Fig. 5.7 (a) shows the basic block graph of this special subproblem: There are five basic blocks $\mathsf{B\_X}$, $\mathsf{B\_XU}$, $\mathsf{B\_\overline{X}D}$, $\mathsf{B\_\overline{X}U}$, and $\mathsf{B\_XD}$ with frequencies $8n$, $4n$, $4n$, $2n$, and $2n$, respectively. In addition, there are three instructions $\mathsf{OP\_X}$, $\mathsf{OP\_\overline{X}U}$, and $\mathsf{OP\_XD}$ with source blocks $\mathsf{B\_X}$, $\mathsf{B\_\overline{X}U}$, and $\mathsf{B\_XD}$, respectively. The candidate blocks of these three instructions are chosen as $\{\mathsf{B\_X}, \mathsf{B\_XU}, \mathsf{B\_\overline{X}D}\}$, $\{\mathsf{B\_XU}, \mathsf{B\_\overline{X}U}\}$, and $\{\mathsf{B\_\overline{X}D}, \mathsf{B\_XD}\}$, respectively.

Figures 5.7 (b) and (c) show two possible schedules with length $1 \cdot 4n + 1 \cdot 2n = 6n$. We observe that *in any optimal schedule* the instruction $\mathsf{OP\_X}$ must have been moved either to block

[5]A further, minor complication resulting from the local dependences is that they introduce redundant edges according to Def. 3.2.7 into the DDG graph—this complicates the structure of the maximal paths and breaks the proof of Theorem 5.1.12.

[6]Interestingly, the problem is polynomial-time solvable if k is chosen as n (2SAT).

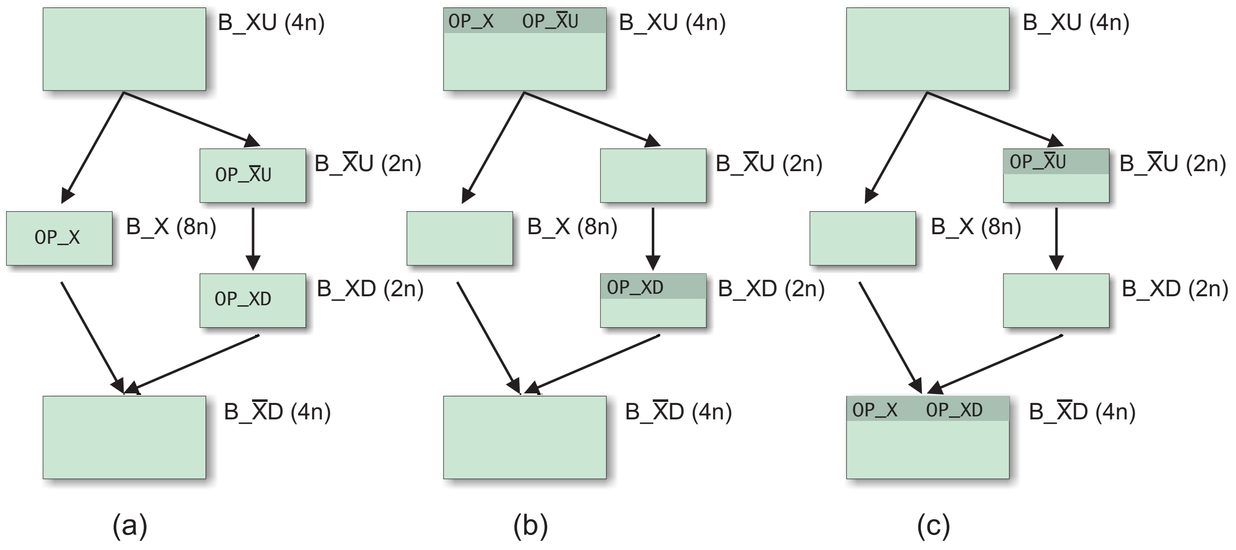

B_XU (4n)
OP_$\overline{\text{X}}$U
B_$\overline{\text{X}}$U (2n)
OP_X
B_X (8n)
OP_XD
B_XD (2n)
B_$\overline{\text{X}}$D (4n)
(a)
OP_X OP_$\overline{\text{X}}$U
B_XU (4n)
B_$\overline{\text{X}}$U (2n)
B_X (8n)
OP_XD
B_XD (2n)
B_$\overline{\text{X}}$D (4n)
(b)
B_XU (4n)
OP_$\overline{\text{X}}$U
B_$\overline{\text{X}}$U (2n)
B_X (8n)
B_XD (2n)
OP_X OP_XD
B_$\overline{\text{X}}$D (4n)
(c)

Figure 5.7: A PCGS flip-flop.

B_XU or B_$\overline{\text{X}}$D (otherwise the schedule length would be at least $8n$). In the former case—and only then—also OP_$\overline{\text{X}}$U must have been moved to B_XU: then it can be scheduled into the same cycle as OP_X "for free", i.e., while not increasing the length of B_XU, and B_$\overline{\text{X}}$U consequently can have length zero (see Fig. 5.7 (b)). The same holds for OP_XD and B_$\overline{\text{X}}$D in the other case (see Fig. 5.7 (c)).

It is evident that there can be no schedules with length less than $6n$. Also, it is clear that there exist no other schedules of this length than the two ones described. Thus these two solely possible optimal solutions demonstrate the ability of this subproblem to act as a "flip-flop" with respect to the block lengths: either $|\text{B\_XU}| = |\text{B\_XD}| = 1$ and $\left|\text{B\_}\overline{\text{X}}\text{D}\right| = \left|\text{B\_}\overline{\text{X}}\text{U}\right| = 0$ or $|\text{B\_XU}| = |\text{B\_XD}| = 0$ and $\left|\text{B\_}\overline{\text{X}}\text{D}\right| = \left|\text{B\_}\overline{\text{X}}\text{U}\right| = 1$ holds. We can relate these two cases to the assignments of a variable: both B_XU and B_XD represent the value of X and B_$\overline{\text{X}}$D and B_$\overline{\text{X}}$U the value of $\neg X$. We will soon see why it is necessary to have *two* blocks representing each true and false assignment of a variable, respectively.

For each of the variables $X_1, \ldots, X_m$ occurring in C, we include one of these subproblems (called flip-flop) in $PCGS(C)$. The index of the variable is inserted into the names of the blocks and instructions to distinguish the different flip-flops.

The next step of the construction connects the m flip-flops to the n clauses of the formula: For each clause C_i we add a further block B_Ci with frequency 1 to the problem which contains two *literal instructions* OP_Li1 and OP_Li2 that represent the literals L_{i1} and L_{i2} in C_i (see Fig. 5.8 (a)). We define that OP_Li2 has a data dependence on OP_Li1 with latency 1. In addition, the block contains a further instruction that can only be scheduled there—the only purpose of this instruction is to ensure that the block has at least length 1 (not shown in the figure).

Now we aim to establish the following relationship: the block of a clause has length 1 in an optimal schedule if and only if the clause is satisfied by the assignment corresponding to the

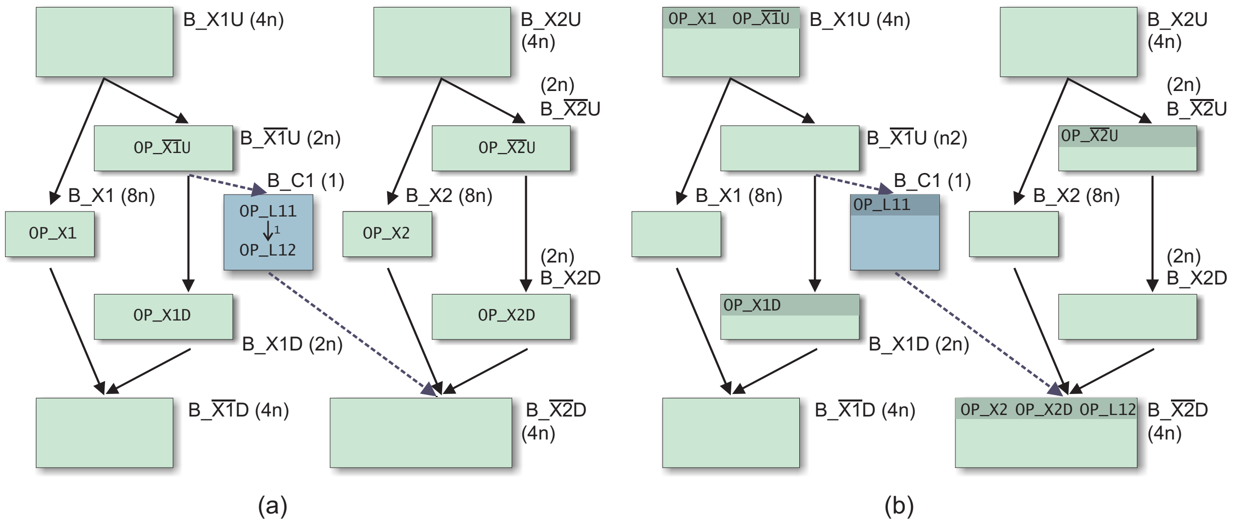


Figure 5.8: Connecting two flip-flops to the block of a clause.

schedule. This is the case if *at least one* literal evaluates to true, or, correspondingly, if *at least one* of the two literal instructions has been moved out of the block—otherwise the block must have length 2 due to the data dependence.

For this purpose, we define B_XjU and B_Ci as the two possible candidate blocks of OP_Li1 (and connect them with a control flow edge) *if* L_{i1} is of the form X_j. Otherwise, if L_{i1} is of the form $\neg X_j$, we designate $\{$B_Ci, B_$\overline{\text{X}j}$U$\}$ as the candidate blocks and add a control flow edge from B_$\overline{\text{X}j}$U to B_Ci. The construction for the second literal L_{i2} is similar: the candidate blocks of OP_Li2 are chosen as $\{$B_Ci, B_XjD$\}$ and $\{$B_Ci, B_$\overline{\text{X}j}$D$\}$ if L_{i2} is of the form X_j and $\neg X_j$ respectively. In general, the instruction of the first literal can only be moved upwards and the instruction of the second one only downwards[7].

Before we give an example, we examine which effect the chosen execution frequencies have on the form of an optimal schedule: Each of the m flip-flops has at least length $6n$ and each of the clause blocks at least length 1, hence $6nm + n \leq T$ holds for the global schedule length T. Moreover, we can always find a schedule with length $T \leq 6nm + 2n$: the schedules of the flip-flops can always be chosen as in Fig. 5.7 (b) or (c), and the n blocks of the clauses have at most length two.

It follows from this upper bound that in any optimal schedule, the block lengths of the flip-flops are either exactly as in Fig. 5.7 (b) or in (c), representing an assignment for the formula (otherwise one flip-flop would have length $8n$ instead of $6n$—then T would be greater than or equal to $6n(m-1)+8n+n = 6mn+3n$). Hence the literal instructions can only be moved into the block of a flip-flop if this block *has length one*—in other words, if the literal *evaluates to one* under the corresponding assignment. And only if at least one literal instruction can be moved out of a clause block can this block *have length one* instead of two—that is, only if at least one

[7]This is why the proof cannot be performed with 3SAT—we have only two directions for code motion.

literal evaluates to one *is the clause satisfied*.

Fig. 5.8 (a) shows an example graph for the clause $\neg X_1 \vee \neg X_2$. Control flow edges from B_$\overline{\text{X1}}$U and to B_$\overline{\text{X2}}$D are connected to the clause block to represent the two literals. The right-hand side (b) depicts an optimal schedule that corresponds to an assignment that assigns true to X_1 and false to X_2. Because block B_$\overline{\text{X2}}$D has length 1 (X_2 is assigned false), instruction OP_L12 can be moved to B_$\overline{\text{X2}}$D (literal $\neg X_2$ evaluates to true) so that block B_Ci has length 1 (the clause C_i is satisfied).

Overall, the relationship between assignments and schedules can be precisely characterized as follows:

- for each *optimal* schedule of $PCGS(C)$ with length $6mn + n + k$ there exists a corresponding assignment for C that satisfies k clauses
- for each assignment that satisfies k clauses there exists a corresponding schedule with length $6mn + n + k$

It follows that an assignment that satisfies k or more clauses exists if and only if an optimal schedule has length $6mn + n + k$ or less. Furthermore, the construction of $PCGS(C)$ can clearly be done in polynomial time. Hence we have reduced Max2SAT to PCGS and with this proven that PCGS is $\mathcal{NP}$-hard. ■

We can conclude from this theorem that the found integral and polynomial sized polytope of PCGS-B is *maximal* in the sense that the inclusion of either the block length or the resource constraints[8] would provably eliminate either its integrality or its polynomial size (if $\mathcal{P} \neq \mathcal{NP}$).

Furthermore, the proof provides some clues *why* adding the block length constraints makes the problem $\mathcal{NP}$-hard: Without these constraints, there are only static maximum lengths for each block given (the numbers of reserved cycles). With block length constraints, however, one can effectively provide via the objective function a maximum value for the *sum* of all block lengths—with the consequence that there can be an exponential number of combinations of individual block lengths such that the sum does not exceed this maximum value. This combinatorial complexity is a common characteristic of $\mathcal{NP}$-hard problems.

Another typical trait concerns the far-reaching "remote effects" of individual decisions: In the same way as the truth assignment of a single variable can synchronously determine the values of literals in many different clauses of a boolean formula, the global movement of an instruction can enforce the synchronous placement of compensation copies of this instruction in many different blocks. This analogy helps explain why global scheduling without resource constraints is $\mathcal{NP}$-complete, but the local variant (without code motion) not.

[8]It is quite clear that the problem becomes $\mathcal{NP}$-complete when the resource constraints are added—details are provided in the next sections.

5.2 Resource-Constrained Global Scheduling

We now concentrate on RCGS, the global scheduling problem without precedence constraints. Earlier work has shown that for both the OASIC and the SILP formulations of local scheduling, the combined polytope of the assignment and resource constraints is polynomial sized and integral [Käs00a]. The local nature of the resource constraints (which operate only on the instructions scheduled at one cycle) suggests that this result can possibly be transferred to the global problem, however, we can—under the assumption $\mathcal{P} \neq \mathcal{NP}$—directly exclude this possibility with the following theorem:

Theorem 5.2.1 *Resource-constrained global instruction scheduling without block length constraints (RCGS-B) is $\mathcal{NP}$-complete.* □

PROOF The proof is similar to the previous one dealing with PCGS, but more straightforward. As in the previous proof, it is clear that RCGS-B is in $\mathcal{NP}$; the main part consists in showing the $\mathcal{NP}$-hardness. This time, we reduce the classic $\mathcal{NP}$-complete problem 3SAT to RCGS-B:

Definition 5.2.2 (3SAT Problem) Given a boolean formula in conjunctive normal form where each clause consists of at most three literals, is there a truth assignment that satisfies the formula? □

Let an instance of this problem be given by a formula $C = C_1 \wedge \ldots \wedge C_n$ where each clause C_i has exactly three literals L_{i1}, L_{i2}, and L_{i3} (otherwise we can duplicate single literals). We will construct an instance of RCGS-B, called $RCGS\_B(C)$, for which a feasible schedule exists if and only if there is a truth assignment that satisfies C. During the construction, we must not use precedence constraints, but we can suppose the existence of resource constraints: we simply assume that only one instruction can be executed per cycle, i.e., there is one slot per cycle. Furthermore, we can define how many cycles should be reserved for each block; these numbers are shown on the left of the blocks in the following figures (one for most blocks, so that they can host not more than one instruction).

As in the proof of Theorem 5.1.25, we use special subproblems, called *flip-flops*, to represent the assignments of variables in the schedule. The flip-flop of a variable x_1, depicted in Fig. 5.9 (a), consists of three blocks and four instructions (displayed in their respective source blocks). Their candidate blocks are defined in such a way that the dummy instructions D_X1 and D_$\overline{\text{X1}}$ cannot be globally moved, but the other *literal instructions* OP_X1 and OP_$\overline{\text{X1}}$ can be moved into the predecessor and successor blocks of their respective source blocks B_X1 and B_$\overline{\text{X1}}$ (which are called *literal blocks*). They *must* also be moved this way since only *one* instruction can be scheduled in these blocks which have one reserved cycle. In particular, at most one can be moved upwards to the block B_X1U; then the other must consequently be moved downwards into each of the successor blocks (in each a compensation copy).

These are the two intended states of the flip-flop: either OP_X1 is moved upwards and OP_$\overline{\text{X1}}$ downwards, representing a true assignment of x_1, or OP_$\overline{\text{X1}}$ is moved upwards and OP_X1 downwards, representing a false assignment. There is a third possibility that represents

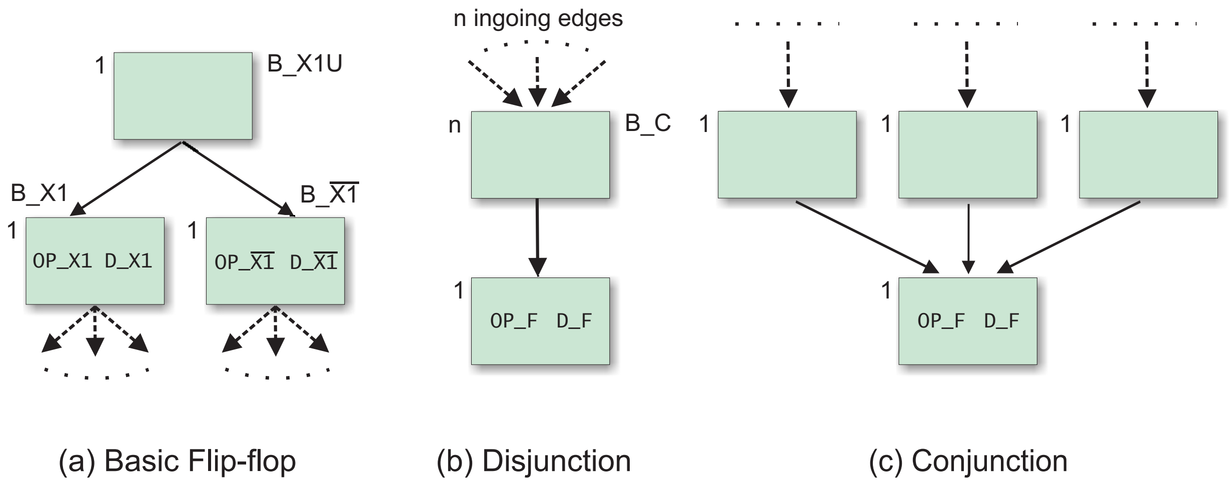


Figure 5.9: Basic elements used to construct $RCGS\_B(C)$.

an undecided state (both instructions moved downwards), but this will turn out to be unproblematic later.

The hereby established relationship between schedules and assignments is that the downward movement of an instruction means that the corresponding literal evaluates to false. Fig. 5.9 shows how conjunctions and disjunctions of literals can be represented under this relationship. The structure of both scheduling subproblems (b) and (c) enforces that, in each feasible schedule, the instruction OP_F is moved upwards (again, this is due to the dummy instruction D_X, which occupies the single reserved slot in its source block). However, this is only possible if each of the successor blocks is not *full*, i.e., has a free slot. This is exploited to model disjunction and conjunction:

- Let there be a *clause* with three literals given. The corresponding scheduling problem consists of a flip-flop for each of the variables occurring in the clause plus the subproblem from Fig. 5.9 (b). Each literal block of a flip-flop, B_Xi and B_$\overline{\text{Xi}}$, is connected to this subproblem via a control flow edge *if* the clause contains the corresponding literal x_i and $\neg x_i$, respectively (at "n ingoing edges", here n=3).
  Then an assignment satisfies the clause if and only if it evaluates less than three literals to false; this corresponds to a schedule where less than three literal instructions are moved downwards into block B_C—then and only then there is at least one free slot in B_C and the instruction OP_F can be moved upwards so that the scheduling problem has a solution.

- Let a boolean formula be given as a *conjunction* of n literals. For the case $n = 3$, these literals are represented by the three upper blocks in Fig. 5.9 (c). As above, we have a flip-flop for each variable and connect each of its two literal blocks, B_Xi and B_$\overline{\text{Xi}}$, to one of these three blocks via a control flow edge if they represent the same literal.
  Then an assignment satisfies the formula if and only if it satisfies *all* n literals; this corresponds to a schedule where *none* of the n literal instructions is moved downwards into

one of the three blocks—then and only then can the instruction OP_F be moved upwards and the scheduling problem has a solution. This is because in each feasible schedule the instruction OP_F must be moved upwards, which means that a compensation copy of this instruction must be scheduled in each of the three upper blocks. This has the consequence that each of them is full and cannot accept another literal instruction.

The problem for the whole formula, $RCGS\_B(C)$, combines these different classes of subproblems: There is a flip-flop for each variable and a clause block for each clause (the upper block of Fig. 5.9 (b) with n=3). These blocks are connected by control flow edges as described above (if a literal does not occur in any of the clauses, then the corresponding literal block has no successor—then we set its maximum length to two instead of one).

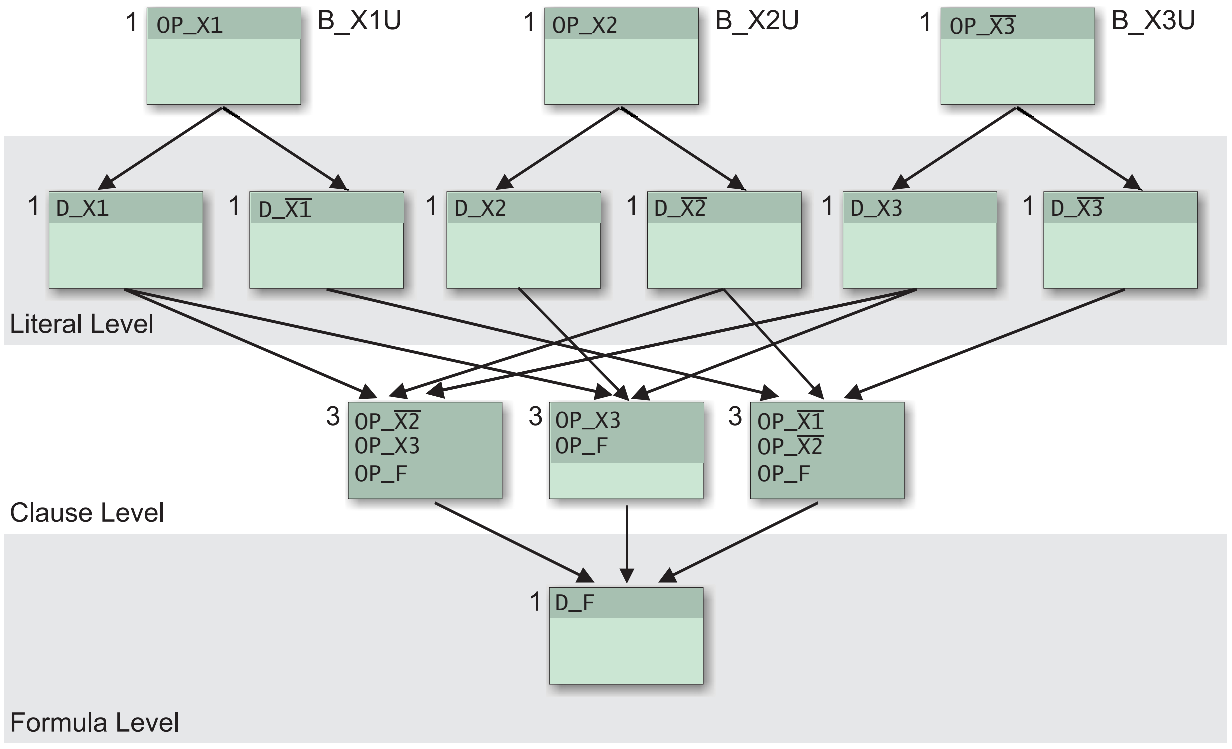


Figure 5.10: Feasible schedule of $RCGS\_B(\widetilde{C})$.

A further block represents the conjunction of all clause blocks—i.e., the formula as a whole. It is the common successor of all clause blocks and has the form of the lower block of Fig. 5.9 (c). A feasible schedule for $RCGS\_B(C)$ then exists if and only if C is satisfiable.

Figure 5.10 depicts a feasible schedule of the RCGS-B problem for the satisfiable formula $\widetilde{C} = (x_1 \wedge \neg x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3)$ as an example. The satisfying truth assignment corresponding to this schedule is $x_1 = 1$, $x_2 = 1$, and $x_3 = 0$. ■

5.2.1 Deriving an Integral Subpolytope of the Resource Constraints

As a consequence of the preceding theorem we now concentrate on the resource constraints alone, which are the same for both local and global scheduling. A common approach models resource binding explicitly in the ILP, i.e., the ILP variables model on which (real) execution unit each instruction is executed [Käs00a, GE93]. This *early resource binding* can be achieved by splitting up each x_n^{At} variable into several new variables $x_n^{At,k_1}, \dots, x_n^{At,k_l}$ that represent the decision to execute on unit $k_1, \dots, k_l$. Let the set of feasible execution unit candidates for each instruction n be given by $R(n)$. Then each occurrence of the variable x_n^{At} is replaced by $\sum_{k \in R(n)} x_n^{At,k}$ in the previously developed constraints[9] and the following resource constraints are added for all execution units k:

$$\sum_{\substack{\forall n: n \in \Theta^{-1}(A) \\ \wedge k \in R(n)}} x_n^{At,k} \leq R_k \qquad \forall A \in \mathcal{B},\ \forall t \in G(A) \tag{5.2.1}$$

Here R_k is equal to one if k represents a single execution unit, but it can be chosen larger if there are multiple equivalent (symmetrical) execution units: these then can be regarded as R_k instances of a type k in order to save variables. The function $\Theta^{-1} : \mathcal{B} \longrightarrow \mathcal{P}(V)$ is defined as the inverse of $\Theta : V \longrightarrow \mathcal{P}(\mathcal{B})$, i.e., $\Theta^{-1}(A) := \{n \in V \,|A \in \Theta(n)\}$.

Example 2 (Early Resource Binding on the Itanium 2) Let n^1_{A}, n^2_{A}, n_{I0}, n^1_{MS}, n^2_{MS}, n_{ML}, $n^1_{\mathsf{CHK.S}}$, and $n^2_{\mathsf{CHK.S}}$ be instructions of the types given by the respective lower indices. If all these instructions can be scheduled in a block A, then the following instances of the resource constraints (5.2.1) are generated for a cycle t:

$$\begin{array}{rcl}
x^{At,\mathsf{M0}}_{n^1_{\mathsf{A}}} + x^{At,\mathsf{M0}}_{n^2_{\mathsf{A}}} + x^{At,\mathsf{M0}}_{n_{\mathsf{ML}}} & \leq & 1 \\
x^{At,\mathsf{M1}}_{n^1_{\mathsf{A}}} + x^{At,\mathsf{M1}}_{n^2_{\mathsf{A}}} + x^{At,\mathsf{M1}}_{n_{\mathsf{ML}}} & \leq & 1 \\
x^{At,\mathsf{M2}}_{n^1_{\mathsf{A}}} + x^{At,\mathsf{M2}}_{n^2_{\mathsf{A}}} + x^{At,\mathsf{M2}}_{n^1_{\mathsf{MS}}} + x^{At,\mathsf{M2}}_{n^2_{\mathsf{MS}}} + x^{At,\mathsf{M2}}_{n^1_{\mathsf{CHK.S}}} + x^{At,\mathsf{M2}}_{n^2_{\mathsf{CHK.S}}} & \leq & 1 \\
x^{At,\mathsf{M3}}_{n^1_{\mathsf{A}}} + x^{At,\mathsf{M3}}_{n^2_{\mathsf{A}}} + x^{At,\mathsf{M3}}_{n^1_{\mathsf{MS}}} + x^{At,\mathsf{M3}}_{n^2_{\mathsf{MS}}} + x^{At,\mathsf{M3}}_{n^1_{\mathsf{CHK.S}}} + x^{At,\mathsf{M3}}_{n^2_{\mathsf{CHK.S}}} & \leq & 1 \\
x^{At,\mathsf{I0}}_{n^1_{\mathsf{A}}} + x^{At,\mathsf{I0}}_{n^2_{\mathsf{A}}} + x^{At,\mathsf{I0}}_{n^1_{\mathsf{CHK.S}}} + x^{At,\mathsf{I0}}_{n^2_{\mathsf{CHK.S}}} + x^{At,\mathsf{I0}}_{n_{\mathsf{I0}}} & \leq & 1 \\
x^{At,\mathsf{I1}}_{n^1_{\mathsf{A}}} + x^{At,\mathsf{I1}}_{n^2_{\mathsf{A}}} + x^{At,\mathsf{I1}}_{n^1_{\mathsf{CHK.S}}} + x^{At,\mathsf{I1}}_{n^2_{\mathsf{CHK.S}}} & \leq & 1
\end{array}$$

A possible resource binding of the instruction group $\{n^1_{\mathsf{A}}, n_{\mathsf{I0}}, n^1_{\mathsf{MS}}, n^2_{\mathsf{MS}}, n_{\mathsf{ML}}, n^1_{\mathsf{CHK.S}}\}$ is represented by $x^{At,\mathsf{M1}}_{n^1_{\mathsf{A}}} = 1$, $x^{At,\mathsf{I0}}_{n_{\mathsf{I0}}} = 1$, $x^{At,\mathsf{M2}}_{n^1_{\mathsf{MS}}} = 1$, $x^{At,\mathsf{M3}}_{n^2_{\mathsf{MS}}} = 1$, $x^{At,\mathsf{M0}}_{n_{\mathsf{ML}}} = 1$, and $x^{At,\mathsf{I1}}_{n^1_{\mathsf{CHK.S}}} = 1$. □

This formulation is not only simple and flexible, but it also produces a highly efficient polytope: [Käs00a] has proven that the resulting constraint matrix is totally unimodular so that these resource constraints form an integral polytope (with Theorem 4.3.5). However, they have the crucial disadvantage that the number of variables needed for an instruction is multiplied by the number of available execution unit type candidates (exhibited by the above example). For highly

[9] Lemma 4.3.14 guarantees that this preserves the integrality of the associated polytopes.

parallel EPIC architectures this multiplies the ILP sizes and, as our experiments indicate, the solution times.

Therefore we have developed in our earlier work [Win01] *hierarchical resource constraints*, which avoid any variable expansion by excluding all resource binding decisions from the ILP model: Not the ILP solver, but the later bundling phase decides on resource binding. The resource constraints only ensure that the binding is later *possible*. We call an instruction group (that is, the set of instructions scheduled at a cycle) *feasible* if a resource binding can be found for it. This requires that the number of instructions does not exceed the number of functional unit candidates (which would constitute a *resource oversubscription*).

This *late resource binding* makes sense anyway since it is the bundling that actually determines the mapping of instructions to execution units on this architecture (see Sec. 2.2.3.2). However, it can raise difficulties if execution or bypass latencies of an instruction depend on the functional unit where it is executed, as on the first-generation Itanium. Fortunately, there are no such interdependences between scheduling and resource binding decisions on the Itanium 2. Nevertheless the resource constraints developed in this chapter can be configured to allow for a combination of both early and late resource binding. They will also lift two further restrictions of the hierarchical resource constraints that are described below.

We first recapitulate our model of the Itanium 2's execution units from Sec. 2.2.1 and refine the notation:

- Each instruction is associated to one of the (execution unit) types, which are sets of those real execution units where the instruction can be executed. Let $\vec{\mathcal{R}}$ be the set of all the execution unit types and $R : V \to \vec{\mathcal{R}}$ be a mapping that associates each instruction to a type. If $R(n) = k$, we often say that instruction n is of type k.

- Let the number of instances of a type be given by the function $c : \vec{\mathcal{R}} \to \mathbb{N}_+$.

The hierarchical resource constraints assume that the types are arranged in a tree-like hierarchy, as depicted in Fig. 2.6. However, this is too rigid for the Itanium 2: There the control speculation checks can be executed on both the MS and I unit types. When modeling this by adding a separate node CHK.S as the common predecessor of these types to the graph, it is no longer a tree.

Moreover, the previous resource constraints assumed that the number of instances of an execution unit type is always equal to the cardinality of the underlying set, i.e., $\forall T \in \vec{\mathcal{R}} : c(T) = |T|$. Yet it can be reasonable to break this rule in order to model issuing-related limitations, for example, it is possible to regard the dispersal window as the most general execution unit type D with $c(\mathsf{D}) = 6$, although it comprises eight real execution units (also shown in Fig. 2.6):

$$\mathsf{D} = \{\mathrm{M0}, \mathrm{M1}, \mathrm{M2}, \mathrm{M3}, \mathrm{I0}, \mathrm{I1}, \mathrm{F0}, \mathrm{F1}\}$$

This removes the need for separate dispersal window constraints with unclear integrality properties as in [Win04, Win01].

Our following new *network flow resource constraints* are more flexible: they employ instead of a tree a more general network for the resource description. Resource binding is regarded as a flow through this network. This might remind of the SILP formulation, which uses a resource flow graph to describe the program execution as a flow of the available execution units through its instructions (see Sec. 8.2.2.1, [Käs00a, Zha96])—however, in our case the instructions flow through the execution units. We first recapitulate the definition and the central theorem of network flow problems [Hag97, NW88]:

Definition 5.2.3 (Network Flow Problem) A *single-source network flow problem (S-NFP)* is given by an acyclic digraph $G = (V, E)$, two nodes s and t called the source and the sink, respectively, and a function $c : E \rightarrow \mathbb{R}$ that assigns to each edge a capacity. A *feasible flow* is a function $f : E \rightarrow \mathbb{R}$ such that:

$$\sum_{\forall u:(u,v)\in E} f(u,v) = \sum_{\forall w:(v,w)\in E} f(v,w) \qquad \forall v \in V \setminus \{s,t\}$$

$$f(e) \leq c(e) \qquad e \in E \tag{5.2.2}$$

The *value* of the flow is defined as $|f| := \sum_{\forall u:(u,t)\in E} f(u,t)$. A flow is called *maximal* if there is no other feasible flow with a greater value. □

Theorem 5.2.4 (Max-Flow Min-Cut Theorem) *The value of the maximum flow of a network flow problem is exactly the minimal capacity of a set of edges to disconnect G with s and t in different components.* □

The proof of the last famous theorem is usually conducted constructively by means of an algorithm that solves the network flow problem; it comprises the following important result:

Theorem 5.2.5 *If all of the edge capacities are integer-valued, then there is a maximum flow that is integer-valued.* □

Now we adapt both the problem definition and the theorem to our purposes. The following modifications are performed:

- We assume *node capacities* instead of edge capacities.
- *Inflow* is allowed not only at the source, but at each node of the graph (this inflow into each node should already be taken into account by the node's capacity limitation).
- There are multiple sinks, namely all those nodes without successors.

Figure 5.11 illustrates how the resulting networks can be employed to model feasible instruction groups: The numbers inside the nodes denote the node capacities; the small horizontal arrows alongside the nodes represent the possibility of inflow. Nodes without this possibility are greyed out to depict that no instructions are associated to these nodes; they are only included for the sake of completeness.

The idea behind this *resource flow network* is that for each feasible instruction group there exists an integer-valued feasible flow and vice versa: The thereby intended correspondence between flows and groups is such that the inflow at each node is equal to the number of instructions

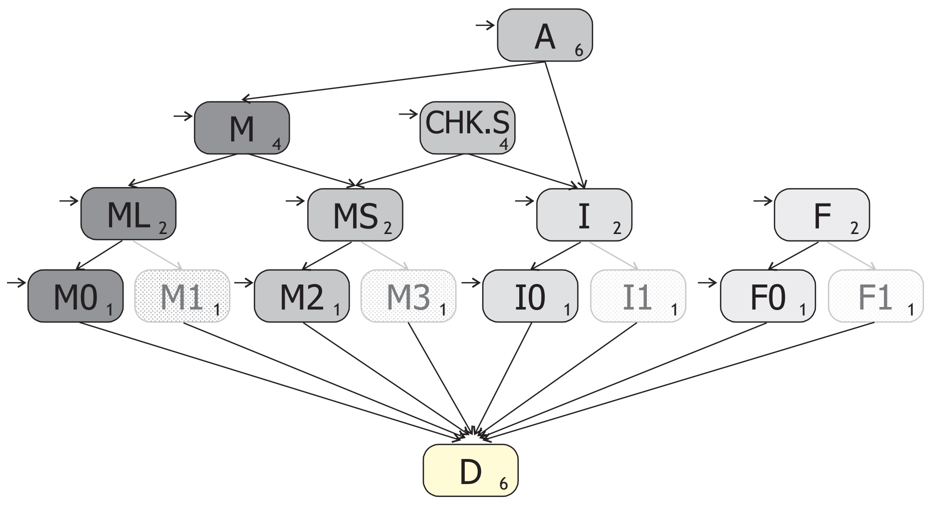


Figure 5.11: Resource flow network for the Itanium 2.

of this type in the group. In the resulting flow, the flow value of each edge represents a number of instructions. The ability of the flow to *fork* at nodes with multiple successors corresponds to the possibility to *decide* between different functional unit types for execution (for example, the inflow at node CHK.S can flow to MS or to I). If an instruction flows through one of the types that represent only one functional unit (in Fig. 5.12 the nodes M0, M1, M2, M3, I0, I1, F0, and F1), then this can be interpreted as binding this instruction to this functional unit.

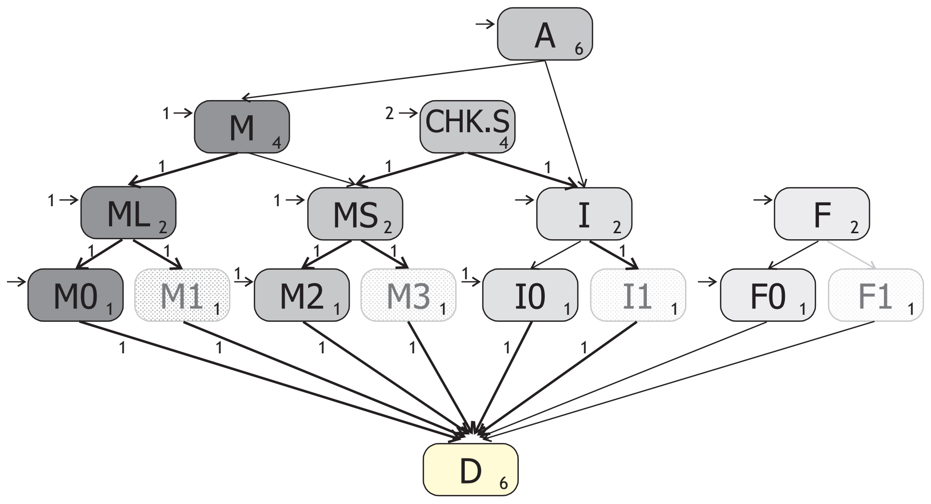


Figure 5.12: Feasible flow in the resource flow network.

Figure 5.12 depicts an example of a flow that represents a feasible instruction group consisting of two speculation checks (type CHK.S), a variable shift (I0), a store (MS), a load (ML), and a floating-point load (M). The inflow is shown to the left of each node (if greater zero). A possible resource binding represented by this flow assigns these instructions to the units M3, I1, I0, M2, M0, and M1, respectively. There are variations possible, for instance it is possible to swap M0 and M1 in the previous assignment. Hence the flows represent resource bindings not completely precisely but leave some ambiguity. The following definitions formalize the described concepts:

Definition 5.2.6 (Multi-Source Network Flow Problem) A *multi-source network flow problem (M-NFP) with node capacities* is given by an acyclic digraph $G = (V, E)$ and a function $c : V \to \mathbb{R}$ that assigns to each node a capacity. An *inflow assignment* is a function $\delta_f : V \to \mathbb{R}$ that gives an inflow value for each node. A *feasible flow* corresponding to this inflow assignment is a function $f : E \to \mathbb{R}$ such that

$$\check{f}(v) = \sum_{\forall w:(v,w)\in E} f(v,w) \qquad \forall v \in V$$

$$\check{f}(v) \leq c(v) \qquad \forall v \in V \tag{5.2.3}$$

where $\check{f}(v)$ is an abbreviation of the flow through node v:

$$\check{f}(v) := \delta_f(v) + \sum_{\forall u:(u,v)\in E} f(u,v)$$

An inflow assignment is said to be *feasible* if there exists a corresponding feasible flow. It is said to be *integral* if $\delta_f(v)$ is integral for all $v \in V$. □

Definition 5.2.7 (Resource Flow Network) A *resource flow network* is a multi-source network flow problem given by

- an acyclic digraph $G_R = (\mathcal{R}, E)$ whose nodes are execution unit types,
- a node capacity function $c : \mathcal{R} \to \mathbb{R}$ that assigns an integral cardinality to each execution unit type,
- and a subset $\vec{\mathcal{R}} \subseteq \mathcal{R}$ of those nodes that permit inflow.

A resource flow network is said to *model* the resource binding of a processor if the following relationship holds: an integral inflow assignment $\delta_f : \vec{\mathcal{R}} \to \mathbb{R}$ is feasible if and only if the instruction group containing $\delta_f(k)$ instructions of functional unit type k is feasible. □

The following overview depicts the relationships between feasible instruction groups, inflow assignments, and flows as established by the two previous definitions:

$$\begin{array}{c} \text{Corresponding} \\ \text{feasible flow} \\ \text{exists} \end{array} \Leftrightarrow \begin{array}{c} \text{Corresponding} \\ \text{inflow assignment} \\ \text{is feasible} \end{array} \Leftrightarrow \begin{array}{c} \text{Instruction} \\ \text{group} \\ \text{is feasible} \end{array} \Leftrightarrow \begin{array}{c} \text{Resource} \\ \text{binding} \\ \text{exists} \end{array}$$

A big advantage of employing network flows in this context is that integral polytopes for related problems are well known [NW88]. Therefore the following theorem is not surprising:

Theorem 5.2.8 (Network Flow Resource Constraints) *Let a resource flow network for the target processor be given as defined previously in Def. 5.2.7. Then the following* network flow resource constraints*, generated* $\forall A \in \mathcal{B}, \forall t \in G(A)$*, form an* integral *polytope of resource constraints:*

$$\sum_{\forall k:(k,l)\in E'} y^{At}_{(k,l)} = \sum_{\forall m:(l,m)\in E'} y^{At}_{(l,m)} \qquad \forall l \in \mathcal{R} \tag{5.2.4}$$

$$\sum_{\forall k:(k,l)\in E'} y^{At}_{(k,l)} \le c(l) \qquad \forall l \in \mathcal{R} \tag{5.2.5}$$

E' contains all edges from E plus a special edge (s,l) for each $l \in \vec{\mathcal{R}}$ with the sole purpose to model the inflow into this node. For each edge $e \in E$ there exists a new integral flow variable *y^{At}_e which holds the value of the flow through this edge; in addition, the* inflow variables *$y^{At}_{(s,l)}$ model the inflow into node l.* □

The functioning of this formulation is evident: it ensures that the total inflow into each node does not exceed its capacity (5.2.5) and is equal to its total outflow (5.2.4) (Kirchhoff's law). The constraint matrix of the equations (5.2.4) is a network matrix and thus totally unimodular [NW88], yielding an integral polytope with Theorem 4.3.5. However, the complete formulation with additional node capacities (5.2.5) goes beyond a network matrix; therefore we provide a separate proof of the theorem in Appendix B.3.1.

In order to connect these network flow resource constraints to the already developed parts of the ILP model, the inflow variables $y^{At}_{(s,l)}$ are for all $l \in \vec{\mathcal{R}}$ replaced by the sum

$$\sum_{\substack{\forall n:n\in\Theta^{-1}(A)\\ \wedge R(n)=l}} x^{At}_n \tag{5.2.6}$$

which is equal to the number of instructions of type l scheduled at cycle t in block A (see Def. 5.2.7). This step does not compromise the integrality of the polytope according to Lemma 4.3.14.

Example 3 (Network Flow Resource Constraints on the Itanium 2) For the instructions from Example 2, the following instances of the network flow resource constraints are generated for a cycle t:

$$\begin{aligned}
x^{At}_{n^1_{\mathsf{A}}} + x^{At}_{n^2_{\mathsf{A}}} - y^{At}_{(\mathsf{A},\mathsf{M})} - y^{At}_{(\mathsf{A},\mathsf{I})} &= 0\\
y^{At}_{(\mathsf{A},\mathsf{M})} - y^{At}_{(\mathsf{M},\mathsf{ML})} - y^{At}_{(\mathsf{M},\mathsf{MS})} &= 0\\
x^{At}_{n^1_{\mathsf{CHK.S}}} + x^{At}_{n^2_{\mathsf{CHK.S}}} - y^{At}_{(\mathsf{CHK.S},\mathsf{MS})} - y^{At}_{(\mathsf{CHK.S},\mathsf{I})} &= 0\\
x^{At}_{n_{\mathsf{ML}}} + y^{At}_{(\mathsf{M},\mathsf{ML})} - y^{At}_{(\mathsf{ML},\mathsf{M0})} - y^{At}_{(\mathsf{ML},\mathsf{M1})} &= 0\\
x^{At}_{n^1_{\mathsf{MS}}} + x^{At}_{n^2_{\mathsf{MS}}} + y^{At}_{(\mathsf{M},\mathsf{MS})} + y^{At}_{(\mathsf{CHK.S},\mathsf{MS})} - y^{At}_{(\mathsf{MS},\mathsf{M2})} - y^{At}_{(\mathsf{MS},\mathsf{M3})} &= 0\\
y^{At}_{(\mathsf{A},\mathsf{I})} + y^{At}_{(\mathsf{CHK.S},\mathsf{I})} - y^{At}_{(\mathsf{I},\mathsf{I0})} - y^{At}_{(\mathsf{I},\mathsf{I1})} &= 0
\end{aligned}$$

$$\begin{aligned}
y^{At}_{(\mathsf{ML},\mathsf{M0})} - y^{At}_{(\mathsf{M0},\mathsf{D})} &= 0 & \qquad y^{At}_{(\mathsf{ML},\mathsf{M1})} - y^{At}_{(\mathsf{M1},\mathsf{D})} &= 0\\
y^{At}_{(\mathsf{MS},\mathsf{M2})} - y^{At}_{(\mathsf{M2},\mathsf{D})} &= 0 & \qquad y^{At}_{(\mathsf{MS},\mathsf{M3})} - y^{At}_{(\mathsf{M3},\mathsf{D})} &= 0
\end{aligned}$$

$$\begin{array}{rclrcl}
x^{At}_{n_{\mathsf{I0}}} + y^{At}_{(\mathsf{I,I0})} - y^{At}_{(\mathsf{I0,D})} & = & 0 & y^{At}_{(\mathsf{I,I1})} - y^{At}_{(\mathsf{I1,D})} & = & 0 \\
x^{At}_{n_{\mathsf{ML}}} + y^{At}_{(\mathsf{M,ML})} & \leq & 2 & y^{At}_{(\mathsf{A,I})} + y^{At}_{(\mathsf{CHK.S,I})} & \leq & 2 \\
 & & & x^{At}_{n^1_{\mathsf{MS}}} + x^{At}_{n^2_{\mathsf{MS}}} + y^{At}_{(\mathsf{M,MS})} + y^{At}_{(\mathsf{CHK.S,MS})} & \leq & 2 \\
y^{At}_{(\mathsf{ML,M0})} & \leq & 1 & y^{At}_{(\mathsf{ML,M1})} & \leq & 1 \\
y^{At}_{(\mathsf{MS,M2})} & \leq & 1 & y^{At}_{(\mathsf{MS,M3})} & \leq & 1 \\
x^{At}_{n_{\mathsf{I0}}} + y^{At}_{(\mathsf{I,I0})} & \leq & 1 & y^{At}_{(\mathsf{I,I1})} & \leq & 1
\end{array}$$

Some of the constraints (5.2.5) need not to be instantiated if it is apparent that the left-hand side is never larger than the right-hand side. Above, the instances for the types A, M, CHK.S, and D could be omitted. □

5.2.2 Reducing the Complexity of the Integral Subpolytope

The above formulation does not multiply the number of x variables like (5.2.1), but it still requires for each cycle modeled in the ILP as many additional flow variables as there are edges in the resource flow network (22 for Fig. 5.11). There are optimizations conceivable how to eliminate many of these auxiliary variables, but the ideal solution would be to remove them completely: As expressed in Def. 5.2.7, it is sufficient to model inflow assignments for which a feasible flow *exists*, but the actual flow itself (i.e., the values of the flow through each edge) is of secondary interest and should be excluded from the ILP.

Thus the remainder of this section will deal with how to remove *all* additional variables from the formulation while keeping it integral. More precisely, our goal is an integral *inflow polytope* that contains the vectors of all feasible inflow assignments. *Node cuts*—minimal sets of nodes that separate parts of the network—will play an important role at this:

Definition 5.2.9 (Node Cut) A *node cut* is a nonempty subset $S \subseteq V$ such that no node in S is dominated or postdominated by other nodes in S (as defined in Def. 3.2.5). A node cut is said to be *complete* if $\mathcal{C}(S) = \mathcal{C}$,[10] i.e., if every complete path in the graph passes through a node of S. Its *capacity* $c(S)$ is the sum of the capacities of its nodes. □

Example 4 In the graph of Fig. 5.11 $S_1 = \{\mathsf{M0}, \mathsf{M1}, \mathsf{MS}, \mathsf{I}\}$ is a node cut, but $S_2 = \{\mathsf{M0}, \mathsf{M1}, \mathsf{M2}, \mathsf{MS}, \mathsf{I}\}$ not (since M2 is dominated by MS) and $S_3 = \{\mathsf{M0}, \mathsf{M1}, \mathsf{MS}, \mathsf{CHK.S}, \mathsf{I}\}$ also not (since CHK.S is postdominated by $\{\mathsf{MS}, \mathsf{I}\}$). □

Lemma 5.2.10 *The following statements are equivalent for a nonempty subset $S \subseteq V$:*

1. *S is a node cut.*
2. *There exists no $S' \subset S$ such that $\mathcal{C}(S') = \mathcal{C}(S)$.* □

[10] The notation $\mathcal{C}(S)$ has been introduced in Sec. 1.3.1 and is used her with respect to the acyclic digraph G_R of the resource flow network.

PROOF (1) $\Rightarrow$ (2): For any node $v \in S$, there exist paths from an entry node to this node and from there to an exit node that do not intersect $S \setminus \{v\}$ (since there exist no nodes in $S \setminus \{v\}$ that dominate and postdominate v, respectively). The concatenation of these paths is a complete path that does traverse S, but not $S \setminus \{v\}$. Hence $c(S') \subsetneq c(S)$ for all $S' \subset S$.

(2) $\Rightarrow$ (1): Let us assume that (2) holds and (1) not, i.e., that S is not a node cut. Then by definition there exists a $v \in S$ that is dominated or postdominated by a set of other nodes $D \subseteq S \setminus \{v\}$. It follows $\mathcal{C}(v) \subseteq \mathcal{C}(D)$ and thus $\mathcal{C}(S \setminus \{v\}) = \mathcal{C}(S)$. Then $S' := S \setminus \{v\}$ is a counter-example for (2). ■

Node cuts can be used to transfer the max-flow min-cut theorem to network flow problems with node capacities instead of edge capacities (the problem from Def. 5.2.3 with Equ. (5.2.2) related to nodes instead of edges):

Theorem 5.2.11 (Max-Flow Min-Cut Theorem (Node Capacities)) *The value of the maximum flow of a network flow problem with node capacities is exactly the minimal capacity of a complete node cut.* □

The proof is fairly obvious (simulate node capacities with edge capacities and apply Theorem 5.2.4), but for the sake of exactness it is given in Appendix B.3.2. The integrality result can also be transferred:

Theorem 5.2.12 *If all of the node capacities are integer-valued, then there is a maximum flow that is integer-valued.* □

As mentioned before, the node cuts are employed to describe the inflow polytope, which contains exactly the (vectors of) feasible inflow assignments. This is based on the following observation: Given a node cut $S \subseteq \mathcal{R}$, $\mathbb{P}_+^{-1}(S)$ refers to those nodes that are postdominated by S in G_R (see Def. 3.2.5). The total inflow into the nodes of $\mathbb{P}_+^{-1}(S)$ is limited by the node cut's capacity, $c(S)$. This is because all inflow into these nodes must eventually flow through S. The following theorem states—as the central result of this section—that this is not only a necessary characterization of feasible flows, but already sufficient:

Theorem 5.2.13 (Inflow Polytope) *Given a resource flow network, any inflow assignment $\delta : \vec{\mathcal{R}} \to \mathbb{R}$ satisfies the following* inflow constraints *if and only if it is feasible:*

$$\sum_{v \in \mathbb{P}_+^{-1}(S)} \delta(v) \leq c(S) \qquad \forall \text{ node cuts } S \subseteq \mathcal{R} \tag{5.2.7}$$

□

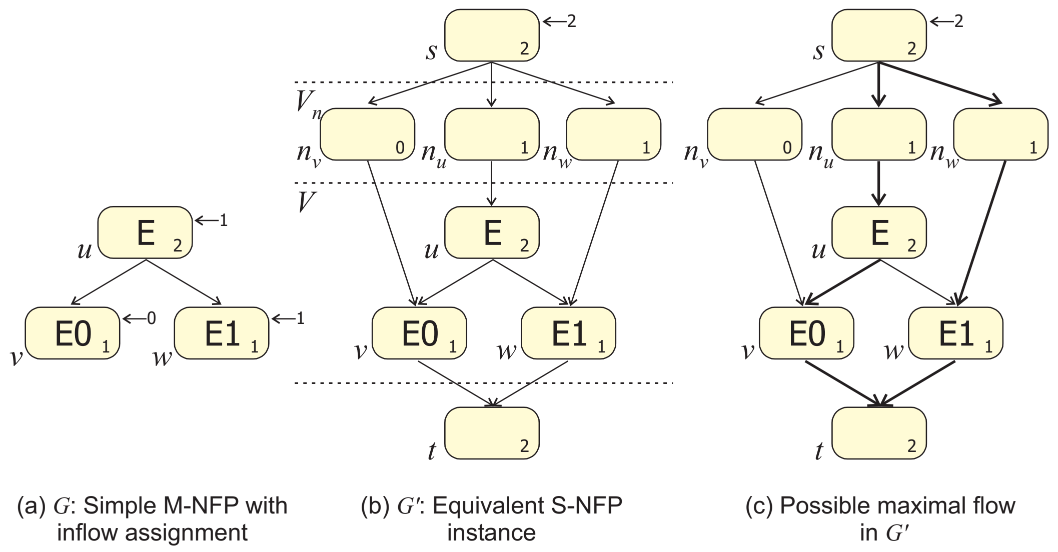
PROOF "$\Rightarrow$": Let an inflow assignment δ be given that satisfies (5.2.7). We employ the backward direction of the max-flow min-cut Theorem 5.2.11 to show that it is feasible, i.e., that a corresponding feasible flow exists. In order to apply this theorem, we first transform the multi-source network flow problem into an equivalent single-source instance $G' = (V', E')$. G' contains G, i.e., we have $E \subseteq E'$ and $V' = V \cup V_n \cup \{s, t\}$ where

- V_n contains for each node $v \in V$ a new node n_v with capacity $c(n_v) := \delta(v)$; hence $c(V_n) = \sum_{v \in V} \delta(v)$. Analogously to $c : \mathcal{P}(\mathcal{R}) \longrightarrow \mathbb{R}$, we define $\delta(V) := \sum_{v \in V} \delta(v)$.

- E' contains all edges from E plus for each $n_v \in V_n$ an edge (n_v, v). The purpose of these edges is to simulate the inflow in the single-source problem (therefore the capacity of each $n_v \in V_n$ is equal to the inflow). In addition, E' contains edges from the source s to all nodes in V_n, and from those nodes in V with no successor in G to the sink t.

- s and t have both capacity $\delta(V)$.

Fig. 5.13 (b) demonstrates the construction of G' for a simple resource flow graph (a) (two execution units E0, E1 belonging to a unit type E). Part (c) shows a possible maximal flow in G' (the bold arcs are those with flow one).



E
2
←1
u
E0
1
←0
v
E1
1
←1
w
(a) G: Simple M-NFP with inflow assignment
s
2
←2
V_n
n_v
0
n_u
1
n_w
1
V
t
2
(b) G′: Equivalent S-NFP instance
(c) Possible maximal flow in G′

Figure 5.13: Example for the construction used in the proof.

From the construction of G' it becomes clear: if there exists a flow f that is feasible in G', then $f\,|_E$ is a feasible flow in G with the same value. In particular, if this value is $\delta(V)$, then the flow is maximal with respect to G' and the inflow at each node in G is exactly $\delta(v)$—the existence of exactly such a flow $f\,|_E$ is to be shown. Hence it is sufficient to prove that there exists a maximal flow with value $\delta(V)$ in G', or—equivalently with the max-flow min-cut Theorem 5.2.11—that the minimal capacity of a complete node cut in G' is $\delta(V)$.

To prove this claim, let such a complete node cut $S \subseteq V'$ be given. Its capacity is $\delta(V)$ in the cases $S = V_n$, $S = \{s\}$, and $S = \{t\}$. For the remaining cases we can partition S into two subsets $S_n = S \cap V_n$ and $S_v = S \cap V$ and estimate:

$$\begin{aligned} c(S) &= c(S_n) + c(S_v) \\ &\overset{\text{Equ. (5.2.7)}}{\geq} c(S_n) + \sum_{v \in \mathbb{P}_+^{-1}(S_v)} \delta(v) \\ &\overset{(*)}{\geq} c(S_n) + \sum_{\substack{\forall u:(u,v)\in E' \\ u \in V_n \setminus S_n}} \delta(v) \\ &\overset{c(n_v)=\delta(v)}{=} c(S_n) + c(V_n \setminus S_n) = c(V_n) = \delta(V) \end{aligned}$$

The step $(*)$ can be performed because every path through $V_n \setminus S_n$ must traverse a node in S_v (otherwise S would not be complete) so that it holds:

$$\{v \in V \,|\exists u : (u,v) \in E' \wedge u \in V_n \setminus S_n\} \subseteq \mathbb{P}_+^{-1}(S_v)$$

Hence there exists a maximal flow f with value $\delta(V)$ in G', and $f\,|_E$ is a feasible flow in G that corresponds to the inflow assignment δ. The existence of such a flow was to be shown.

”$\Leftarrow$”: Let δ be feasible so that a corresponding feasible flow exits. Let f denote such a flow. It has to be demonstrated that Equ. (5.2.7) holds for any node cut S. Since all nodes in $\mathbb{P}_+^{-1}(S)$ are postdominated by S, all inflow into these nodes must also flow through S due to flow conservation, hence we can estimate:

$$\sum_{v \in \mathbb{P}_+^{-1}(S)} \delta(v) \leq \sum_{v \in S} \check{f}(v) \overset{\text{Equ. (5.2.3)}}{\leq} c(S)$$

■

An important addition to the above theorem is that a feasible flow exists that is integral. This is expressed by the subsequent corollary; it follows directly by applying Theorem 5.2.12 to G' in the above proof.

Corollary 5.2.14 *For any integral inflow assignment $\delta : \vec{\mathcal{R}} \to \mathbb{R}$ that satisfies the inflow constraints (5.2.7) there exists a corresponding feasible flow that is integral.* □

This corollary confirms that the idea behind the resource flow networks is sound, namely that the flow values represent numbers of instructions (which must naturally be integers). An integral inflow assignment for which only corresponding flows exist that are *non-integral* would contradict this intuition.

The inequalities (5.2.7) describe the inflow polytope if the term $\delta(v)$ is replaced by $y^{At}_{(s,v)}$:

$$\sum_{v \in \mathbb{P}_+^{-1}(S)} y^{At}_{(s,v)} \leq c(S) \qquad \forall \text{ node cuts } S \subseteq \mathcal{R} \tag{5.2.8}$$

We denote one of these *inflow constraints*, instantiated for a node cut S, as h_S. We can easily prove that they describe an integral polytope:

Theorem 5.2.15 *The inflow polytope is integral.* □

PROOF The inflow polytope is exactly the *projection* of the integral polytope of the network flow resource constraints from Theorem 5.2.8 onto the subspace composed of all those vectors in which all components belonging to the flow variables y_e^{At} ($\forall e \in E$) are equal to zero: There exists a vector in this polytope (a feasible flow) if and only if there exists a vector with the same inflow components in the inflow polytope. Notably, we have proven this relationship in Theorem 5.2.13 above for *real-valued* inflow assignments, as required for the equivalence of polytopes. With Lemma 4.3.13, the projection is integral. ■

For any given resource flow network that models the resource binding of a processor, the inflow polytope delivers integral resource constraints without additional variables. However, it is necessary to find all the node cuts in the resource flow network in order to generate all the inflow constraints. The number of these constraints for a single scheduling position is unclear, but likely exponential (an analysis follows below). Although it grows only with the size of the resource flow network and not with the program size, it is necessary to generate this number of constraints for each scheduling position modeled in the ILP (in other words, for each of the reserved cycles of each block)—thus it is not negligible.

In Sec. 5.1.2 we have already successfully reduced an exponential sized formulation to a polynomial sized one by removing redundant constraints. The following theorem identifies two sources of redundancy in the inflow constraints:

Theorem 5.2.16 *A subset $\dot{\mathcal{H}}$ of constraints is redundant in the description of the inflow polytope if for each constraint $h_S \in \dot{\mathcal{H}}$ one of the following two cases applies:*

1. *there exist node cuts S_1 and S_2 such that $S_1 \cap S_2 = \emptyset$, $S_1 \cup S_2 = S$, and $\mathbb{P}_+^{-1}(S_1) \cup \mathbb{P}_+^{-1}(S_2) = \mathbb{P}_+^{-1}(S)$. S is called* atomic *if such S_1 and S_2 do not exist.*
2. *there exists a node cut S' such that $\mathbb{P}_+^{-1}(S) \subset \mathbb{P}_+^{-1}(S')$ and $c(S) \geq c(S')$. S is called* tight *if such an S' does not exist.* □

PROOF We apply Lemma 4.3.3 to prove the redundancy of the subset. We use as $\ll$ the transitive closure of the following relation $\blacktriangleleft \subseteq \dot{\mathcal{H}} \times \dot{\mathcal{H}}$:

- If S is redundant due to the first case, we define $h_{S_1} \blacktriangleleft h_S$ if $h_{S_1} \in \dot{\mathcal{H}}$ and $h_{S_2} \blacktriangleleft h_S$ if $h_{S_2} \in \dot{\mathcal{H}}$.
- If S is redundant due to the second case, we define $h_{S'} \blacktriangleleft h_S$ if $h_{S'} \in \dot{\mathcal{H}}$.

In order to build the transitive closure, we must ensure that there exists no sequence $h_{S_1} \blacktriangleleft h_{S_2} \blacktriangleleft \dots \blacktriangleleft h_{S_k}$ such that $h_{S_1} = h_{S_k}$. For this we observe that for each $i \in \{1, \dots k-1\}$, $h_{S_i} \blacktriangleleft h_{S_{i+1}}$ holds in this sequence either due to the first or the second case defined above—then either $c(S_i) < c(S_{i+1})$ or $c(S_i) \leq c(S_{i+1})$ holds.

It follows $c(S_1) < c(S_k) \Rightarrow S_1 \neq S_k$ if there is at least one $i \in \{1, \ldots k-1\}$ such that $h_{S_i} \blacktriangleleft h_{S_{i+1}}$ holds due to the first case. Otherwise we have $\mathbb{P}_+^{-1}(S_i) \supset \mathbb{P}_+^{-1}(S_{i+1})$ for all $i \in \{1, \ldots k-1\}$ and thus $\mathbb{P}_+^{-1}(S_1) \supsetneqq \mathbb{P}_+^{-1}(S_k) \Rightarrow S_1 \neq S_k$. Thus the transitive closure $\ll$ does exist and imposes an irreflexive, antisymmetric, and transitive relation—a strict partial order—on $\dot{\mathcal{H}}$, as required by Lemma 4.3.3.

Now the proof of the actual redundancy is straightforward. In the first case, the sum of h_{S_1} and h_{S_2} yields exactly h_S, with $h_{S_1} \ll h_S$ and $h_{S_2} \ll h_S$ if $h_{S_1} \in \dot{\mathcal{H}}$ and $h_{S_2} \in \dot{\mathcal{H}}$, respectively. In the second case, $h_{S'}$ subsumes h_S and $h_{S'} \ll h_S$ if $h_{S,} \in \dot{\mathcal{H}}$. Thus, with Lemma 4.3.3 all constraints from $\dot{\mathcal{H}}$ can be removed from the description of the inflow polytope. ■

The *first statement* of the above theorem says that a constraint is redundant if it is subsumed by the sum of two smaller constraints. For example, $S = \{\mathsf{M0}, \mathsf{M1}, \mathsf{M2}\}$ is not atomic in Fig. 5.11 since it can be partitioned into two subsets $S_1 = \{\mathsf{M0}, \mathsf{M1}\}$ and $S_2 = \{\mathsf{M2}\}$ such that $\mathbb{P}_+^{-1}(S_1) \cup \mathbb{P}_+^{-1}(S_2) = \{\mathsf{ML}, \mathsf{M0}, \mathsf{M1}\} \cup \{\mathsf{M2}\} = \{\mathsf{ML}, \mathsf{M0}, \mathsf{M1}, \mathsf{M2}\} = \mathbb{P}_+^{-1}(S)$. In contrast, for the node cut $S' = \{\mathsf{M0}, \mathsf{M1}, \mathsf{M2}, \mathsf{M3}\}$ there does not exist such a partition, since M is postdominated by S', but not by any proper subset of S'.

The *second statement* says that a constraint is redundant if its node cut can be "moved downwards" in the graph (figuratively) without increasing its capacity. For instance, in Fig. 5.11 $S = \{\mathsf{ML}, \mathsf{MS}\}$ can be extended to $S' = \{\mathsf{M0}, \mathsf{M1}, \mathsf{M2}, \mathsf{M3}\}$ with the same capacity, and $\mathbb{P}_+^{-1}(S) = \{\mathsf{M}, \mathsf{ML}, \mathsf{MS}\} \subset \{\mathsf{M}, \mathsf{ML}, \mathsf{MS}, \mathsf{M0}, \mathsf{M1}, \mathsf{M2}, \mathsf{M3}\} = \mathbb{P}_+^{-1}(S')$.

We call an inflow constraint (5.2.8) *basic* if it is created for an atomic and tight node cut. The constraint created for $S' = \{\mathsf{M0}, \mathsf{M1}, \mathsf{M2}, \mathsf{M3}\}$ is an example of this. It follows inductively from Theorem 5.2.16 that each inflow constraint is subsumed by a linear combination of basic inflow constraints. This result can be extended to linear combinations of constraints:

Corollary 5.2.17 *For each linear combination of inflow constraints there exists a linear combination of basic inflow constraints that subsumes it.* □

Corollary 5.2.18 *All basic inflow constraints constitute a description of the inflow polytope.* □

It is apparent that the confinement to basic constraints allows us to remove a significant number of inequalities from the formulation. Nevertheless, an example can be found where an exponential number remains:

Example 5 The resource flow network in Fig. 5.14 describes, for a given k, $3k+1$ execution unit types. The $2k$ nodes $\mathsf{E}i_\mathsf{a}$ and $\mathsf{E}i_\mathsf{b}$ can be interpreted as real execution units; for each of these units, there exists a class of instructions that can only be executed there (visible from the associated inflow arrows). Alternatively, one of the $\mathsf{E}i_\mathsf{a}$ and $\mathsf{E}i_\mathsf{b}$ units can accept one instruction of the more general types $\mathsf{E}i$ or E.

This example is artificial, but not unrealistic, since there are various factors that could explain the limited number of $\mathsf{E}i$ and E instructions (such as common utilization of a certain subunit or a bus, or even encoding restrictions). It demonstrates the expressiveness of the network flow resource constraints, but also its potential complexity:

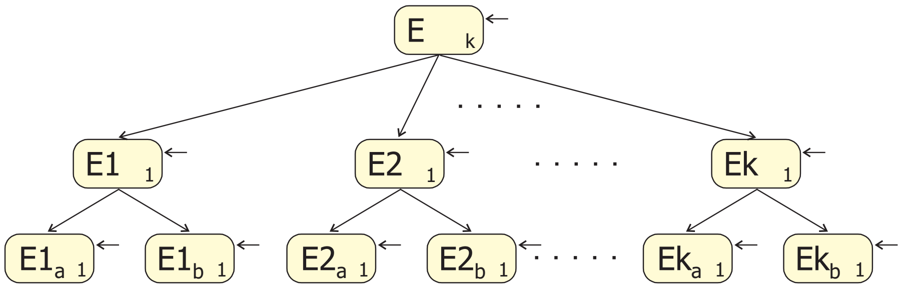


Figure 5.14: Resource flow network with an exponential number of basic inflow constraints.

We can assign to each sequence $a \in \{0,1\}^k$ a node cut S_a that contains E*i* if $a_i = 1$ and both E*i*a and E*i*b else. S_a is obviously tight and it is atomic since $\mathsf{E} \in \mathbb{P}_+^{-1}(S_a)$, but $\mathsf{E} \notin \mathbb{P}_+^{-1}(S)$ for any proper subset $S \subset S_a$. Hence there exist at least 2^k basic inflow constraints. □

The consequences of this example are not immediately clear: It is still possible that we have not yet discovered all the redundancy in the formulation and that the number of integral facets of the inflow polytope is in reality polynomial. However, an indication against this is that in the above example we can construct for each inflow constraint an infeasible inflow assignment that violates only this constraint. The following theorem shows that this holds even in the general case, so that "basic" is synonymous to "irredundant":

Theorem 5.2.19 (Minimality) *All basic inflow constraints constitute a* minimal *description of the inflow polytope.* □

It is necessary to show for any basic constraint the *non-existence* of a linear combination of other inflow constraints that subsumes it. The proof of this is extensive (as typical for non-existence proofs) and therefore given in Appendix B.3.3.

Thus the worst-case complexity remains prohibitive in the general case, but it turns out to be easily manageable in the case of the Itanium 2: There are a total of 11 atomic and tight node cuts in the resource flow network of Fig. 5.11, namely:

$$\mathcal{I} = \{\{\mathsf{M0}\}, \{\mathsf{M2}\}, \{\mathsf{I0}\}, \{\mathsf{F0}\}, \{\mathsf{M0}, \mathsf{M1}\}, \{\mathsf{M2}, \mathsf{M3}\}, \{\mathsf{I0}, \mathsf{I1}\}, \{\mathsf{F0}, \mathsf{F1}\} \\ \{\mathsf{M0}, \mathsf{M1}, \mathsf{M2}, \mathsf{M3}\}, \{\mathsf{M2}, \mathsf{M3}, \mathsf{I0}, \mathsf{I1}\}, \{\mathsf{D}\}\}$$

Analogously to the previous network flow resource constraints, the inflow constraints (5.2.8) are adapted to the model by replacing the inflow variables $y_{(s,l)}^{At}$ on their left-hand sides by (5.2.6). They are termed *network inflow resource constraints*. Given the above set of node cuts, $\mathcal{I}$, eleven of them are instantiated for each $A \in \mathcal{B}$ and each cycle $t \in G(A)$:

$$\sum_{k \in \mathbb{P}_+^{-1}(S)} \sum_{\substack{\forall n: n \in \Theta^{-1}(A) \\ \wedge R(n)=k}} x_n^{At} \leq c(S) \qquad \forall S \in \mathcal{I} \tag{5.2.9}$$

An instance of this inequality must only be added to the ILP if the number of binary variables on the left-hand side exceeds the value on the right-hand side—otherwise it is evidently redundant. Hence we need *at most* 11 inequalities per scheduling position and *no additional variables* to model the resource constraints of the Itanium 2. The subpolytope described by these inequalities is *integral* with Theorem 5.2.15 and Lemma 4.3.14. This allows to conclude that the found formulation is highly efficient for the resource flow network of the Itanium 2. The following example demonstrates this, especially in comparison to the earlier formulations in Examples 2 and 3:

Example 6 (Network Inflow Resource Constraints) For the instructions from Example 2, the following instances of the network flow resource constraints need to be instantiated for a cycle t:

$$\begin{array}{rcl} x^{At}_{n^1_{\mathsf{MS}}} + x^{At}_{n^2_{\mathsf{MS}}} + x^{At}_{n^1_{\mathsf{CHK.S}}} + x^{At}_{n^2_{\mathsf{CHK.S}}} + x^{At}_{n_{\mathsf{I0}}} & \leq & 4 \\ x^{At}_{n^1_{\mathsf{A}}} + x^{At}_{n^2_{\mathsf{A}}} + x^{At}_{n_{\mathsf{ML}}} + x^{At}_{n^1_{\mathsf{MS}}} + x^{At}_{n^2_{\mathsf{MS}}} + x^{At}_{n^1_{\mathsf{CHK.S}}} + x^{At}_{n^2_{\mathsf{CHK.S}}} + x^{At}_{n_{\mathsf{I0}}} & \leq & 6 \end{array} \qquad \square$$

Another advantage of the network inflow resource constraints is the high flexibility and expressiveness of the resource flow network: it clearly encompasses earlier description formalisms like the hierarchical resource constraints in [Win01] and the resource graph in [Käs00a, Zha96]. In addition, it allows to model more general issuing limitations, which can be due to the common use of subunits or due to encoding restrictions (like the dispersal window, which cannot be modeled using the two formalisms mentioned earlier).

However, the question arises how to deal with the exponential worst-case complexity, which possibly materializes as a result of more comprehensive networks with unfavorable structures. This inevitable complexity comes from the fact that we project the polynomial sized polytope of Theorem 5.2.8 onto a lower-dimensional plane. It is not uncommon that an exponential number of facets emerge in the projection [AZ96]; in Sec. 5.1.2, we have encountered this effect in the opposite direction: there we have lifted an exponential sized polytope to a higher-dimensional space in order to obtain a polynomial sized formulation.

A way to contain the possible complexity increase is to perform only a partial projection. A detailed discussion of this, however, would go beyond the scope of this thesis. A more realistic adaptation that could become necessary on other target microarchitectures is a *partial reintroduction* of early resource binding to allow for variable latencies, i.e., latencies that vary with the unit type an instruction is executed on. For example, if an instruction n can be executed on the unit types $k_1, \ldots, k_l \in \vec{\mathcal{R}}$ with different latencies, then a partial early binding can be modeled by splitting up each x^{At}_n variable into l new variables $x^{At,k_1}_n, \ldots, x^{At,k_l}_n$ that represent the decisions to execute on the respective unit types. These variables are incorporated into the model as described at the beginning of Sec. 5.2.1; the precedence constraints related to n, however, must differentiate between the l unit types to allow for the different latencies. Such resource-binding-aware precedence constraints have been presented in [Win01]. In the inflow resource constraints (5.2.9), the l variables are treated as the x^{At} variables of l different imaginary instructions of the types $k_1, \ldots, k_l$ so that the integrality of the inflow polytope is not affected.

5.3 Refinement and Summary of the Model

Before we provide a summary of the entire ILP model for global scheduling (PCGS $\cap$ RCGS), we will revert the modifications we have applied to the given scheduling region to meet the five requirements of Remark 5.1.1. We have already dealt with requirement (4)—the remaining ones determine, inter alia, that the candidate blocks of instructions comprise all predecessors and successors of their source block—possibly including the newly added JS blocks, which have no counterparts in the original problem.

As discussed already in Sec. 5.1, these artificial candidate block ranges are unrealistic and may impair the correctness: Instructions should only be scheduled in their original, effective candidate blocks as introduced by Def. 3.3.2. Let $\Theta^x(n) \subseteq \Theta(n)$ denote these blocks[11], then we can replace the x_n variables of all blocks in $\Theta(n) \setminus \Theta^x(n)$ by zero in all constraints of the ILP formulation—namely in Equ. (5.1.15), (5.1.16), (5.1.17), and (5.2.9). In doing so, we remove the possibility to schedule instructions into these blocks from the ILP model, as intended.

This replacement only excludes scheduling decisions by *predetermining* some of the variable values—the polytope remains correct and also integral with Corollary 4.3.12. We can additionally eliminate all x variables this way that result from the increased values of $\mathbb{G}_A$ due to Remark 5.1.1-(2). All these adaptations will be incorporated below in the final ILP model of global scheduling. There we will also remove redundant a variables and, as a consequence of this, redundant constraints. For this, we need the following precise specification of candidate blocks. Starting with this definition, the notion "candidate blocks" is meant to refer again to the original, effective candidate blocks from Def. 3.3.2, denoted by $\Theta^x(n)$.

Definition 5.3.1 (Valid Candidate Block Range) A *valid range of candidate blocks* for an instruction $n \in V$ is given by four sets with the following properties: The set $\Theta(n)$ denotes the *potential candidate blocks*; it remains the same as in Remark 5.1.1-(1)

$$\Theta(n) = \mathcal{B}^{\preceq}(s(n)) \cup \mathcal{B}^{\succeq}(s(n)) \tag{5.3.1}$$

and is partitioned into two subsets:

$$\Theta(n) = \Theta^x(n) \cup \Theta^{\setminus x}(n)$$

$\Theta^x(n)$ contains the *(actual) candidate blocks* for which both x_n and a_n variables are generated. $\Theta^{\setminus x}(n)$ denotes those blocks for which no x_n variables are produced. This set includes Ω and all JS blocks. It can be further partitioned into those blocks for which only a_n variables exist ($\Theta^{a\setminus x}(n)$) and those blocks for which neither sort of variables is instantiated ($\Theta^-(n)$):

$$\Theta^{\setminus x}(n) = \Theta^{a\setminus x}(n) \cup \Theta^-(n)$$

$\Theta^a(n) := \Theta^x(n) \cup \Theta^{a\setminus x}(n)$ combines those blocks that have a_n variables. The following table summarizes the different subsets of $\Theta(n)$:

[11] Where $n \in V$ is, as often implicitly supposed in this section, an arbitrary given instruction.

		$\Theta^{\setminus x}$	
	Θ^a		
	Θ^x	$\Theta^{a\setminus x}$	Θ^-
a variables exist	$+$	$+$	$-$
x variables exist	$+$	$-$	$-$

We require that the set of candidate blocks, $\Theta^x(n)$, fulfills the following three properties:

1. $\forall(m,n) \in E_D$:
$$A \in \Theta(m) \cap \mathcal{B}^\frown(\Theta^x(m)) \Rightarrow A \notin \Theta^x(n) \tag{5.3.2}$$
$$A \in \Theta(n) \cap \mathcal{B}^\smile(\Theta^x(n)) \Rightarrow A \notin \Theta^x(m) \tag{5.3.3}$$
In these formulas, $\mathcal{B}^\frown(\mathcal{A})$ and $\mathcal{B}^\smile(\mathcal{A})$ denote those blocks in G_B that are located before and after *all* blocks in $\mathcal{A} \subseteq \mathcal{B}$, respectively:
$$\mathcal{B}^\frown(\mathcal{A}) := \{A \in \mathcal{B} \,|\, \neg\exists D \in \mathcal{A} : D \preceq A\} \qquad \mathcal{B}^\smile(\mathcal{A}) := \{A \in \mathcal{B} \,|\, \neg\exists D \in \mathcal{A} : D \succeq A\}$$

2. For each BBG subpath $(A, C_1, \ldots, C_k, B) \in \mathcal{B}^+$ between two blocks $A, B \in \Theta^x(n)$ holds:
$$\{C_1, \ldots, C_k\} \subseteq \Theta^{\setminus x}(n) \Rightarrow (\mathcal{B}^\succ(A) \cap \mathcal{B}^\prec(B)) \subseteq \Theta^{\setminus x}(n) \tag{5.3.4}$$

3. The set $\Theta^a(n)$ is *contiguous* in G_B. A subset $\mathcal{B}' \subseteq \mathcal{B}$ is called contiguous in G_B if it includes all blocks that lie on a path in G_B between any two blocks in it:
$$A, B \in \mathcal{B}' \Rightarrow \left(\mathcal{B}^\succeq(A) \cap \mathcal{B}^\preceq(B)\right) \subseteq \mathcal{B}' \tag{5.3.5}$$

□

The definition allows to specify blocks in $\Theta^{\setminus x}(n)$ for which no a_n variables are produced. This allows for the fact that the set of candidate blocks of an instruction n, $\Theta^x(n)$, is in practice often small and contains sometimes only a single block. Then it is unnecessary and wasteful to instantiate a_n variables for all predecessors and successors of these blocks in $\Theta(n)$—these variables must have the same values anyway and can be eliminated. We ignore this possibility at the moment and will investigate later how the removal can take place.

The first two required properties are easy to fulfill by any scheduling region—they basically say that a block can be removed from the range of candidate blocks of an instruction if the latter *cannot* be scheduled there due to scheduling constraints. This removal of superfluous candidate blocks does not affect the correctness, but it simplifies proofs and reduces the ILP sizes.

The **first property** states that, if n is dependent on m, then on any program path through the BBG the candidate block range of m must not start later or end later than the range of n—this is because copies of m must always be scheduled before those of n. For example, suppose that Equ. (5.3.2) is violated, i.e., that there is a block $A \in \Theta^x(n) \cap \Theta(m)$ such that neither A nor one of its predecessors are elements of $\Theta^x(m)$. From the proof of Proposition 5.1.6 we know that there exists a path $C \in \mathcal{C}(s(m)) \cap \mathcal{C}(s(n))$ that passes through A. If a copy of n is scheduled in A, then a copy of m must be scheduled in A or in a predecessor of A on C—but this is not possible due to the choice of $\Theta^x(m)$. So no copy of n can be scheduled in A without violating the precedence constraints, thus we can remove A from $\Theta^x(n)$.

The **second property** expresses that if the inner blocks on a path between two blocks $A, B \in \Theta^x(n)$ have no x_n variables, then the values of the x_n variables along any other path between the two blocks must all be zero, too. Thus these variables can also be removed (to minimize $\Theta^x(n)$). Formally, this is a consequence of the following corollary:

Corollary 5.3.2 *For each instruction $n \in V$ and two blocks $A, B \in \Theta(n)$, $A \prec B$, holds: in any feasible solution of the global scheduling ILP, the sum of the x_n variables of the inner blocks on any subpath between A and B has the same value.* □

PROOF Follows from Lemma 5.1.19: the value it refers to is $a_n^{\uparrow B} - \left(a_n^{\uparrow A} + \sum_{t \in G(A)} x_n^{At}\right)$. ■

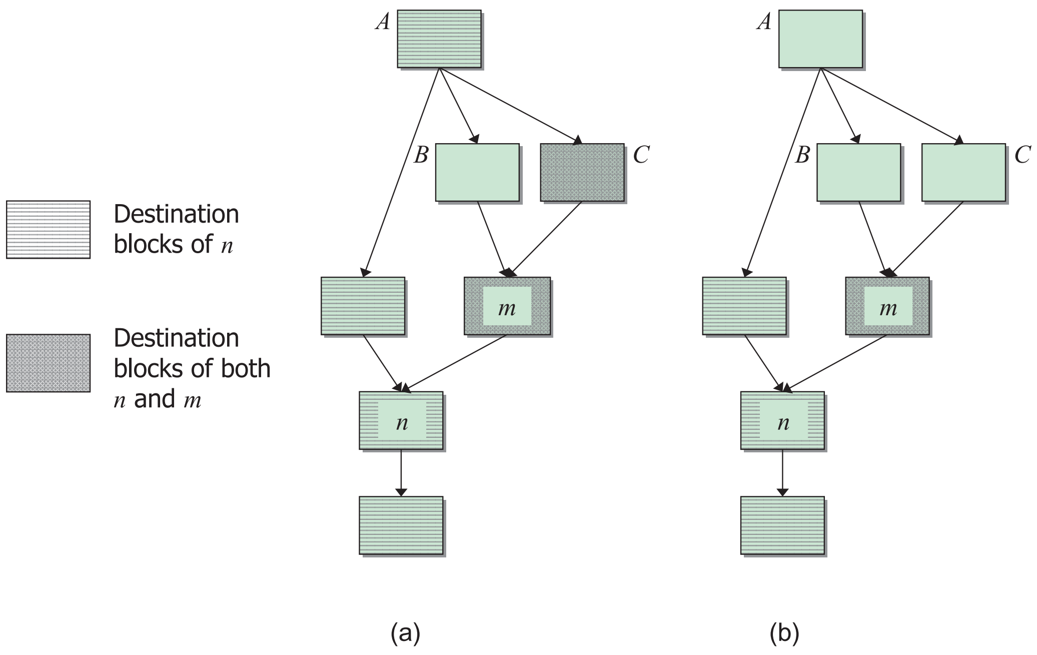

Destination blocks of n
Destination blocks of both n and m
A
B
C
m
n
(a)
(b)

Figure 5.15: Candidate blocks $\Theta^x(m)$ and $\Theta^x(n)$ before and after enforcement of the properties of Def. 5.3.1.

A small example in Fig. 5.15 demonstrates how the enforcement of the two properties can reduce the candidate block ranges: In (a), the two dependent instructions are shown in their respective source blocks; the initial candidate block ranges are depicted by different patterns. We assume that m is non-speculative so that the range of candidate blocks for this instruction is limited. We furthermore suppose that B is a JS block and thus no candidate block. This has the consequence that Equ. (5.3.4) removes block C from both $\Theta^x(m)$ and $\Theta^x(n)$ (via the path (A, B, D)). Since $(m, n) \in E_D$ exists, Equ. (5.3.2) then removes A from $\Theta^x(n)$.

The **third property** in Def. 5.3.1 says that the blocks with a_n variables in the set $\Theta^a(n)$ form contiguous regions without "holes". This condition helps to keep the ILP model correct and simple at the same time, also with regard to later extensions. By definition, $\Theta^a(n)$ is a superset of the set $\Theta^x(n)$, which itself does not have to be contiguous; in fact, we have reserved in Sec. 3.3.2 the possibility to exclude individual blocks from $\Theta^x(n)$ arbitrarily. We choose $\Theta^a(n)$ as the *smallest contiguous set* enclosing $\Theta^x(n)$, namely as:

$$\Theta^a(n) := \Theta(n) \setminus \big(\mathcal{B}^\frown(\Theta^x(n)) \cup \mathcal{B}^\smile(\Theta^x(n))\big)$$

This choice implies $\Theta^-(n) = \Theta(n) \cap \big(\mathcal{B}^\frown(\Theta^x(n)) \cup \mathcal{B}^\smile(\Theta^x(n))\big)$; it is based on the observation that the a_n variables of the blocks in $\Theta(n) \cap \mathcal{B}^\frown(\Theta^x(n))$ and $\Theta(n) \cap \mathcal{B}^\smile(\Theta^x(n))$ must have the value zero and one, respectively, *in any feasible solution*. This can be seen from each program path $(A_1, \ldots, A_k, \Omega) \in \mathcal{B}^+$ through $s(n)$: If $\{A_1, \ldots, A_i\} \subseteq \Theta^-(n) \cap \mathcal{B}^\frown(\Theta^x(n))$ and $\{A_j, \ldots, A_k, \Omega\} \subseteq \Theta^-(n) \cap \mathcal{B}^\smile(\Theta^x(n))$ denote all those blocks along the path that are elements of the respective sets ($i < j$), then any a-x constraint (5.1.16) generated for a BBG edge $(A, B) \in E_B$ such that $A, B \in \{A_1, \ldots, A_i\}$ or $A, B \in \{A_j, \ldots, A_k, \Omega\}$ has the form:

$$a_n^{\uparrow B} = a_n^{\uparrow A}$$

Since Equ. (5.1.16) instantiated for $A_1 \in \Theta^-(n)$ yields $a_n^{\uparrow A_1} = 0$ and Equ. (5.1.17) $a_n^{\uparrow \Omega} = 1$, it follows inductively from these equations that the values of all a_n variables of the blocks in $\{A_1, \ldots, A_i\}$ and $\{A_j, \ldots, A_k, \Omega\}$ must be equal to zero and one, respectively, in any feasible solution. Therefore we can *replace* these variables by the constants zero and one, respectively, in the ILP formulation. This substitution decreases the dimension of the polytope while preserving its integrality: The latter follows from the observation that the new polytope is the projection of the old one onto the subspace of all those points with $A \in \Theta^-(n) \cap \mathcal{B}^\frown(\Theta^x(n)) \Rightarrow a_n^{\uparrow A} = 0$ and $A \in \Theta^-(n) \cap \mathcal{B}^\smile(\Theta^x(n)) \Rightarrow a_n^{\uparrow A} = 1$ (Lemma 4.3.13).

The substitution renders many ILP constraints redundant: To begin with, instances of the precedence constraints (5.1.15) created for an $A \in \Theta^-(n) \cap \mathcal{B}^\frown(\Theta^x(n))$ and an $A \in \Theta^-(m) \cap \mathcal{B}^\smile(\Theta^x(m))$ have the property that the first two and the last two terms on their left-hand side must be zero, respectively, so that they are always satisfied and thus redundant. If $A \in \Theta^-(n) \cap \mathcal{B}^\smile(\Theta^x(n))$, then it must also hold $A \in \Theta^-(m) \cap \mathcal{B}^\smile(\Theta^x(m))$ due to Equ. (5.3.3), hence such instances are redundant, too. Analogously, $A \in \Theta^-(m) \cap \mathcal{B}^\frown(\Theta^x(m))$ implies $A \in \Theta^-(n) \cap \mathcal{B}^\frown(\Theta^x(n))$ with Equ. (5.3.2) so that we have shown altogether the redundancy of any precedence constraints generated for an $A \in \Theta^-(m) \cup \Theta^-(n)$. Hence it is sufficient to instantiate them for blocks in $\Theta^a(m) \cap \Theta^a(n)$.

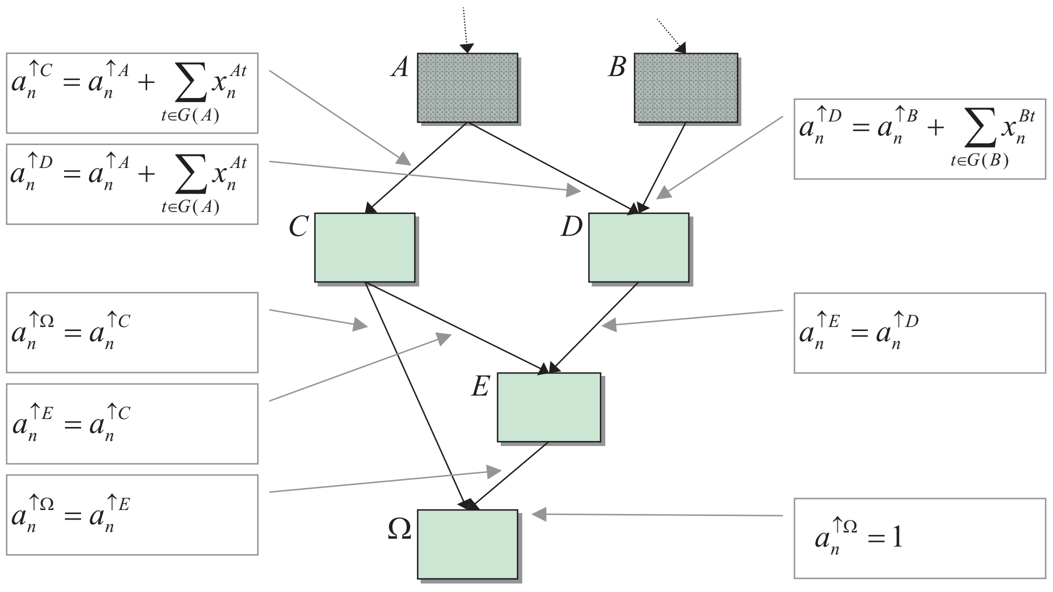
The a-x constraints (5.1.16) created for block pairs from $\Theta^-(n) \cap \mathcal{B}^\frown(\Theta^x(n))$ and $\Theta^-(n) \cap \mathcal{B}^\smile(\Theta^x(n))$ have the form $0 = 0$ and $1 = 1$, respectively, after the substitution and can be omitted, too. As shown by the later Equ. (5.3.11), this omission occurs by instantiating a-x-constraints only for the blocks $B \in \Theta^a(n)$, however, this unintendedly also leaves out the following instances generated for an edge $(A, B) \in E_B$ such that $A \in \Theta^a(n)$ and $B \in \Theta^-(n) \cap \mathcal{B}^\smile(\Theta^x(n))$:

$$1 = a_n^{\uparrow A} + \sum_{t \in G(A)} x_n^{At} \tag{5.3.6}$$

These equations are added separately for all $A \in \widetilde{\Theta^a}(n)$, which are those block such that there exists an edge $(A, B) \in E_B$ as described above:

$$\widetilde{\Theta^a}(n) := \left\{A \in \Theta^a(n) \,\middle|\, \exists B \in \Theta^-(n) : (A, B) \in E_B\right\} \tag{5.3.7}$$

Equ. (5.3.6) can be regarded as the replacement for the constraints (5.1.17).



$a_n^{\uparrow C} = a_n^{\uparrow A} + \sum_{t\in G(A)} x_n^{At}$
$a_n^{\uparrow D} = a_n^{\uparrow A} + \sum_{t\in G(A)} x_n^{At}$
$a_n^{\uparrow \Omega} = a_n^{\uparrow C}$
$a_n^{\uparrow E} = a_n^{\uparrow C}$
$a_n^{\uparrow \Omega} = a_n^{\uparrow E}$
A
B
C
D
E
Ω
$a_n^{\uparrow D} = a_n^{\uparrow B} + \sum_{t\in G(B)} x_n^{Bt}$
$a_n^{\uparrow E} = a_n^{\uparrow D}$
$a_n^{\uparrow \Omega} = 1$

Figure 5.16: Example of redundant a_n variables.

Figure 5.16 shows an example of the variable and constraint removal (the blocks in $\Theta^-(n) \cap \mathcal{B}^\smile(\Theta^x(n))$ are shown without pattern, those in $\Theta^a(n)$ with pattern). The originally generated assignment constraints are given along with the BBG edges; they are *all* removed. Instead, instances of (5.3.6) are added for all blocks in $\widetilde{\Theta^a}(n) = \{A, B\}$. The variables $a_n^{\uparrow C}$, $a_n^{\uparrow D}$, $a_n^{\uparrow E}$ and $a_n^{\uparrow \Omega}$ occur no longer in the new formulation.

The following theorem shows that we can go even a step further and restrict the instantiation of precedence constraints to blocks in $\Theta^a(m) \cap \Theta^x(n)$:

Theorem 5.3.3 *A precedence constraint (5.1.15) is redundant and can be omitted if* $A \notin \Theta^x(n)$.□

PROOF Consider a constraint (5.1.15) produced for a dependence $(m, n) \in E_D$ and a block $A \in \Theta(m) \cap \Theta^{\setminus x}(n)$; we show that it is redundant. There exists a path $C \in \mathcal{C}(s(m)) \cap \mathcal{C}(s(n))$ through A as shown in the proof of Proposition 5.1.6. If *no* predecessor of A on C is element of $\Theta^x(n)$, then the first two terms of the left-hand side of Equ. (5.1.15) must be zero (the $a_n^{\uparrow A}$ and the left double sum)—but then the constraint is always satisfied and can be omitted.
Otherwise, let $A_1, A_2, \ldots$ be the blocks we encounter when moving along C from A upwards ($A_1 = A$), and let A_k be the first encountered predecessor that is element of $\Theta^x(n)$. Due to

Equ. (5.3.2) this block must also be element of $\Theta^a(m)$ so that precedence constraints are generated for it and $(m, n) \in E_D$. We call such a constraint instantiated for $t := \mathbb{G}_A - w_{mn} + 1$ the "A_k constraint" and observe that it subsumes the inequality

$$a_n^{\uparrow A_{k-1}} + (1 - a_m^{\uparrow A_{k-1}}) \leq 1 \tag{5.3.8}$$

(see Equ. (5.1.20) for details). We show that this inequality itself subsumes the constraint generated for A (referred to as the A constraint): The value of $a_n^{\uparrow A_{k-1}}$ must be equal to the sum of the first two terms in the A constraint since no x_n variables exist for all the blocks $A_1, \ldots, A_{k-1}$. The term $(1 - a_m^{\uparrow A_{k-1}})$, however, is greater or equal to $(1 - a_m^{\uparrow A})$ (since the values of the a variables are according to Prop. 5.1.24 always monotonically decreasing), which itself is greater or equal to the last two terms of the A constraint. Thus the A_k constraint is tighter than the A constraint and renders the latter redundant. ■

Eventually, we now revert the third and last remaining requirement of Remark 5.1.1, namely that the scheduling region must be free of JS edges. We show that the ILP model is—after a small modification—independent of whether JS edges are removed or not so that the requirement becomes dispensable. For this purpose, we look at differences in the generated constraints with and without JS edge removal:

Let a JS edge (A, B) in the original BBG be given that is split up into two new edges (A, J) and (J, B) to and from a newly added JS block J, respectively. Since $\forall n \in V : J \notin \Theta^x(n)$ (Def. 5.3.1), the insertion of the JS block has only then an effect on the model at all if $A \in \Theta^a(n)$ and $B \in \Theta^a(n)$ for an instruction n (otherwise $\forall n \in V : J \in \Theta^-(n)$). Thus we assume $A, B \in \Theta^a(n)$ and consider the two cases $J \in \Theta^a(n)$ and $J \notin \Theta^a(n)$ separately:

If $J \in \Theta^a(n)$, then the a-x constraint (5.1.16) generated for $(J, B) \in E_B$ has the form $a_n^{\uparrow B} = a_n^{\uparrow J}$. If the JS edge (A, B) is *not* removed, the resulting model is the same except that this equation is missing and that $a_n^{\uparrow B}$ replaces all occurrences of $a_n^{\uparrow J}$. Since these two variables have same value anyway, this does not affect the correctness or the integrality properties of the polytope.

The case $J \notin \Theta^a(n)$ can only occur if the edge (A, B) goes from a predecessor to a successor of $s(n)$ (*bypassing* $s(n)$)—then $J \notin \Theta(n)$, i.e., it is neither predecessor nor successor of $s(n)$. If in this case the JS edge is not split up, then falsely a-x constraints for the BBG edge (A, B) are added (which must not occur because this edge is not included in any program path in $\mathcal{C}(s(n))$, cf. Equ. (5.1.11)). To prevent this from occurring, we must *exclude* the instantiation of these constraints for all such bypassing BBG edges, collected in the following set:

$$E_n^{\times} = \{(A, B) \in E_B \,|A \prec s(n) \wedge s(n) \prec B\} \tag{5.3.9}$$

As an example, Fig. 5.17 shows an excerpt of a BBG where the JS edge (A, B) bypasses the source block of n and must be ignored during the generation of constraints related to instruction. This omission is shown in Equ. (5.3.11) and (5.3.12) of the following listing which recapitulates all constraints of the developed ILP model, also allowing for the other modifications developed in this section.

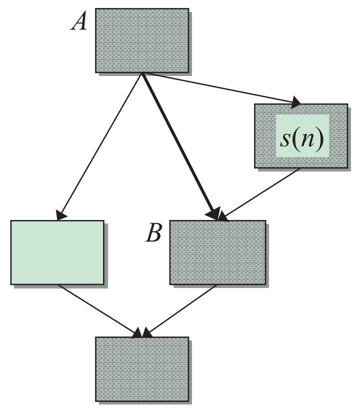


Figure 5.17: Example of a bypassing JS edge; the candidate blocks of instruction n are shown with pattern.

In the block length constraints, the B_t^A variables correspond—as explained in Sec. 5.1.2—to the $x_{l_A}^{At}$ variables of an imaginary instruction l_A that is scheduled once in each block (Equ. (5.3.15)), but not earlier than any other instruction there. The constraints (5.3.14) are the local precedence constraints that allow for this dependence of l_A on all other instructions scheduled into the block. The additional variable B_0^A is equal to one if and only if the block A is empty in the schedule (in any solution that is optimal under an objective function that minimizes $B_1^A, \ldots, B_{\mathbb{G}_A}^A$).

. . . .

Summary of the Developed ILP Model

Assignment Constraints:

$$a_n^{\uparrow A} + \sum_{t \in G(A)} x_n^{At} = 1 \qquad \forall n \in V,\, \forall A \in \widetilde{\Theta^a}(n) \tag{5.3.10}$$

$$a_n^{\uparrow B} = a_n^{\uparrow A} + \sum_{t \in G(A)} x_n^{At} \qquad \forall n \in V,\, \forall B, A \in \Theta^a(n) : (A, B) \in E_B \setminus E_n^{\times} \tag{5.3.11}$$

The sums in both equations are omitted if $A \notin \Theta^x(n)$. The right-hand side of an a-x constraint (5.3.11) is replaced by zero if for a $B \in \Theta^a(n)$ no block A with the described property exists.

($\mathcal{O}\left(|V| \cdot |E_B|\right)$ instances)

Precedence Constraints:

$$a_n^{\uparrow A} + \sum_{\substack{t_n \in G(A) \\ t_n \leq t + w_{mn} - 1}} x_n^{At_n} + \sum_{\substack{t_m \in G(A) \\ t_m \geq t}} x_m^{At_m} + (1 - a_m^{\uparrow B}) \leq 1 \tag{5.3.12}$$

$$\forall (m,n) \in E_D,\, \forall A \in \Theta^a(m) \cap \Theta^x(n),\, \forall t \in G_{mn}(A),\; B \in \Theta^a(m) : (A,B) \in E_B \setminus E_m^{\times}$$

The second double sum is omitted if $A \notin \Theta^x(m)$. As indicated by the missing universal quantifier, instances of the inequality need only to be created for one (arbitrary) block B with the described property (if missing, the term $(1 - a_m^{\uparrow B})$ is replaced by zero).

($\mathcal{O}\left(\mathbb{G} \cdot |E_D|\right)$ instances)

Resource Constraints:

$$\sum_{k \in \mathbb{P}_+^{-1}(S)} \sum_{\substack{\forall n : n \in \Theta^{x^{-1}}(A) \\ \wedge R(n) = k}} x_n^{At} \leq c(S) \qquad \forall S \in \mathcal{I},\, \forall A \in \mathcal{B},\, \forall t \in G(A) \tag{5.3.13}$$

Instances with $c(S)$ or fewer x variables on the left-hand side are redundant and can therefore be omitted.

($\mathcal{O}\left(\mathbb{G} \cdot |\mathcal{I}|\right)$ instances)

Block Length Constraints:

$$\sum_{0 \leq t_n < t} B_{t_n}^A + \sum_{t \leq t_m \leq \mathbb{G}_A} x_m^{At_m} \leq 1 \qquad \forall A \in \mathcal{B},\, \forall m \in \Theta^{x^{-1}}(A),\, \forall t \in G(A) \tag{5.3.14}$$

$$\sum_{t=0}^{\mathbb{G}_A} B_t^A = 1 \qquad \forall A \in \mathcal{B} \tag{5.3.15}$$

($\mathcal{O}\left(\mathbb{G} \cdot |V|\right)$ instances)

Objective Function:

$$\min \sum_{A \in \mathcal{B}} f_A \cdot \left(\sum_{t \in G(A)} t \cdot B_t^A \right) \tag{5.3.16}$$

. . . .

We can assume $\mathbb{G} = \mathcal{O}(|V|)$, $|\mathcal{B}| = \mathcal{O}(|V|)$ and that $|\mathcal{I}|$ is fixed. Then the asymptotic overall complexity of the formulation is $\mathcal{O}\left(|V|^2\right)$ variables (including $\mathcal{O}(\mathbb{G} \cdot |V|)$ x variables, $\mathcal{O}(|\mathcal{B}| \cdot |V|)$ a variables) and $\mathcal{O}(\mathbb{G} \cdot |E_D|) = \mathcal{O}\left(|V|^3\right)$ constraints. This shows that the value $\mathbb{G}$ heavily affects the sizes and thereby the solution times of the produced ILPs. Thus $\mathbb{G}_A$ should be chosen as small as possible for each basic block A. However, the ILP solver could choose to grow less frequently executed blocks by moving code into them—this possibility should not be limited by a too small $\mathbb{G}_A$.

A safe choice is to collect all instructions that could possibly be moved into the block, $\Theta^{x^{-1}}(A)$, and compute via list scheduling an upper bound on the length of an optimal local schedule of all these instructions. A more realistic possibility is to set $\mathbb{G}_A$ heuristically to the product (or the sum) of a constant factor and the original length of the block. This factor could be chosen inversely proportionally to the execution frequency, f_A, to allow for increased code motion into colder blocks. We will deal with this issue again in the description of the implementation in Sec. 7.1.

The above summary concludes the development of our ILP model for global instruction scheduling. Before we continue with extensions in the next chapter, we recapitulate the proven complexity results. The results of Theorems 5.1.25 and 5.2.1 can be summarized as follows:

Corollary 5.3.4 *Global instruction scheduling is already $\mathcal{NP}$-complete even without the resource or precedence constraints.* □

Strikingly, this corollary does not apply to local scheduling where the $\mathcal{NP}$-completeness disappears if either resource or precedence constraints are removed. Hence it highlights a wide complexity gap between the local and the global variant. As remarked in the discussion of the $\mathcal{NP}$-completeness proofs, this inherently higher complexity can be partly attributed to the necessity to schedule compensation copies synchronously in different basic blocks. This multiplies the interactions between scheduling decisions.

Fig. 5.18 provides a more detailed overview. It contrasts the complexities of several subproblems of global and local scheduling: either they are $\mathcal{NP}$-hard or an integral and polynomial sized subpolytope is known to exist. The results on the right-hand side are due to [Käs00a, Win01]; regarding the polyhedral complexity of local scheduling without resource constraints, however, it should be noted that the integrality proofs in both of these works do not comprise the block length constraints. But since this problem is trivially in $\mathcal{P}$,[12] we can *presume* that an integral and

[12]Schedule each instruction n at the earliest possible cycle, $ASAP(n)$—the resulting schedule has minimal length.

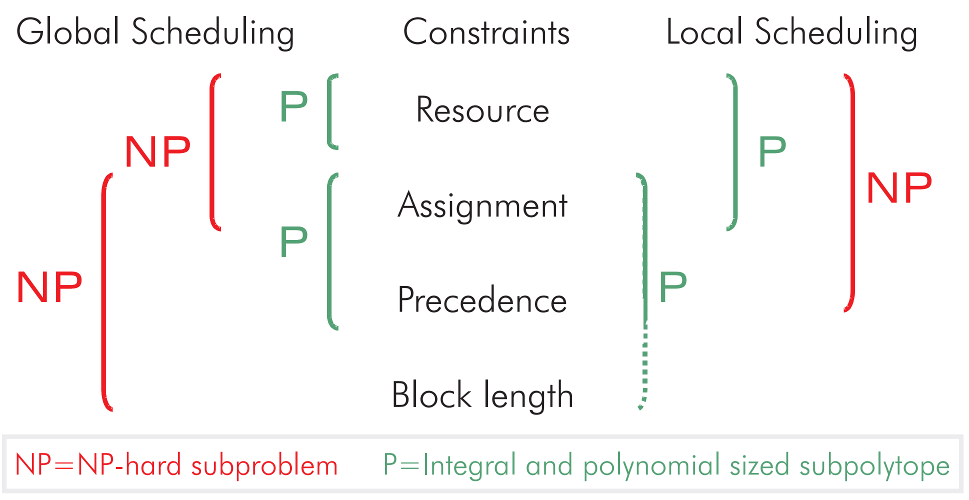


Figure 5.18: Complexity overview.

polynomial sized polytope does exist (thus the dotted brace). The results on the left-hand side are from Theorems 5.1.25, 5.2.1 and 5.2.8 and from Corollary 5.1.23.

Two points are remarkable in Fig. 5.18: Firstly, as already stated by the corollary, subproblems that are polynomial for local scheduling are $\mathcal{NP}$-hard for global scheduling. Secondly, the integral and polynomial sized subpolytopes of global scheduling are *maximal* in the sense that they cannot be extended by other constraint classes without reaching $\mathcal{NP}$-completeness. This substantiates that the found formulation is close to maximal efficiency.

Chapter 6

Extensions of the Model

The ILP model developed in the previous chapter is our starting point for several extensions dealing with predication, speculation, and additional kinds of code motion. We take care that the extended ILP formulation remains *correct* and as *efficient* as possible, but we do not maintain the same level of formality in the presentation as in the previous chapter.

Completeness of the extensions is not the foremost goal in this chapter, but practicability: sometimes we impose restrictions where more flexibility would not seem promising or introduce too much additional complexity. We favor lightweight extensions over exhaustive ones. Otherwise we would also run into danger of overstraining the available ILP solver technology—as it will be seen in Sec. 7.3.1, the combined complexity even of the current extensions adds already significantly to the overall solution times of the model.

6.1 Predication, Branches, and If-Conversion

Predication can be used to extend the scope of code motion for non-speculative instructions, as it will be shown in the following. We cannot expect that the given input program is free from predicated instructions: Apart from conditional branches—which will be dealt with at the end of this section—we can suppose that possibly if-conversion has already been applied in clear cases, or that expressions have been directly translated into predicated instruction sequences by the code selector.

We can also suppose that these *originally predicated instructions* are inherently non-speculative: the fact that a predicate register controls and restricts their execution is an integral part of their semantics—executing them speculatively would be practically equivalent to removing this qualifying predicate register[1]. Furthermore, we can assume that BBG edges have predicate registers associated with them as described in Def. 3.2.3; we also suppose that these registers are unique, possibly as a result of renaming. However, we also allow for the possibility that such registers might not be available for some fall-through edges[2] (depending on the input program)—then we mark these edge by a void predicate register `pV`.

[1]In fact, this can be done as an optimization under certain conditions. Details are provided in Sec. 6.2.1.

[2]I. e., the control flow edge to the subsequent block in memory if no branch is taken at the end of a basic block.

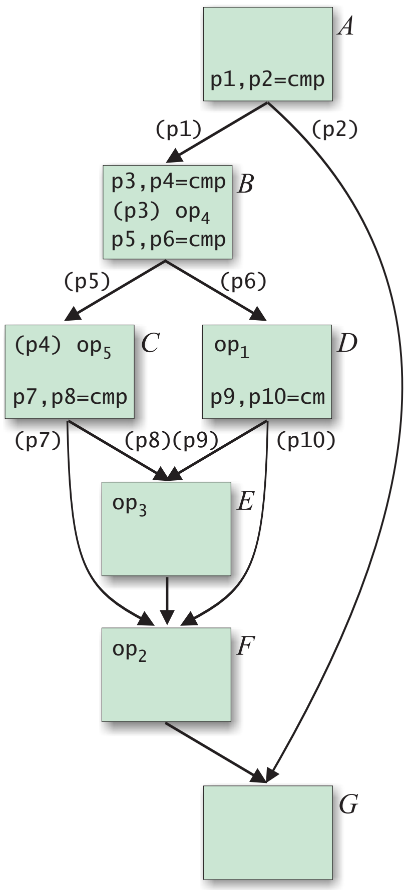


Figure 6.1: Case study of predication. All instructions $\mathtt{op}_1$-$\mathtt{op}_5$ are considered non-speculative.

We say that an instruction is *controlled by a predicate register*, or short *predicate-controlled*, if during the execution of the input program the instruction is executed if and only if this predicate register is true. A closer look shows that not every predicated instruction is predicate-controlled: In Fig. 6.1,[3] for instance, the instruction $\mathtt{op}_5$ is executed only if *both* `p4` and `p5` are true—but there is no predicate register that holds $\mathtt{p4} \wedge \mathtt{p5}$. The difference comes from the fact that our notion of "predicate-controlled" is global (in order to be useful for global code motion): $\mathtt{op}_5$ is controlled by `p4` only locally in block C, but not within the whole scheduling region. The qualifying predicate is only then also the controlling predicate of an instruction if its source block is control equivalent to that of the compare (as is the case with $\mathtt{op}_4$). To find the controlling predicate of an originally unpredicated instruction, we examine the *postdominance frontier* of its source block (cf. [SS02], here defined as an edge set):

[3]In this and in all other case studies in this chapter, only a small number of instructions are shown (those that are relevant). They are given in a general, non-specific form "$\mathtt{op}_n$ `rDEST=rSOURCE` " (sometimes abbreviated "`rDEST=rSOURCE`").

Definition 6.1.1 (Postdominance (Edge) Frontier) The postdominance frontier of a block D, $\widetilde{\mathbb{P}_+}(D) \subseteq E_B$, is the set of edges (A, B) such that B is postdominated by D and A not. $\square$

If $\widetilde{\mathbb{P}_+}(s(n)) \neq \emptyset$, then at runtime the control reaches the source block of an instruction n if and only if it flows along one of the edges in this set. Hence, if the latter contains only one edge, then the predicate register associated with this edge is a controlling predicate. For instance, this set contains only (A, B) for $\texttt{op}_2$, hence `p1` controls this instruction.

If a controlling predicate is available for a non-speculative instruction, and if the instruction is made data dependent on the compare that generates it (the *controlling compare*), then the restrictions with respect to code motion due to its non-speculativeness no longer apply—it can then be scheduled into speculative destination blocks as long as it is there guarded by the controlling predicate register to eliminate the speculativeness (*predicated code motion* [SRM+94]). That way it is *scheduled* speculatively, but never *executed* speculatively. It can be moved arbitrarily far upwards and downwards—like a speculative instruction—as long as the RAW dependence on the compare is preserved (and this compare itself is *not* speculated). In doing so, we incorporate predication into the ILP *as a side effect of speculative code motion*.

However, the destination block candidates of downward code motion might be limited by the availability of the controlling predicate: If the destination block is not dominated by the compare's source block, then there exists a path to it where it is undefined. For example, if in Fig. 6.1 $\texttt{op}_1$ is moved to G and guarded by `p6` there, then this predicate register is not written at all if the path A-G is taken. This can be prevented by initializing the register to zero at a point that dominates all candidate blocks (block A is such a point in the example). For this, the instruction `cmp.ne.and p1,p2=0,r0` can be scheduled that clears both destination predicate registers. A larger selection of predicate registers can be written via the instruction `mov pr=r0,mask`.[4]

A controlling predicate does not necessarily exist for every instruction. For instance, it is not possible to determine one in the above way if the postdominance frontier contains more than one edge. But then we can try to find *partially controlling predicates* as follows: We successively mark, for all $(A, B) \in \widetilde{\mathbb{P}_+}(s(n))$, the block A and all of its predecessors with the predicate register associated with this BBG edge (including possibly `pV`). If during this procedure blocks are marked more than once, we remove the marking permanently (we do not mark them again).

The candidate block range $\Theta^x(n)$ can then be extended by all blocks that are marked after this procedure (if at the same time dependence edges from the controlling compares are added): if at runtime the control flows through a block marked by a predicate register that is associated with an edge $e \in \widetilde{\mathbb{P}_+}(s(n))$, then it flows through e and $s(n)$ *if and only if* if this register has value one. Thus the instruction can be scheduled speculatively there if guarded by the predicate register.

In Fig. 6.1, for example, $\widetilde{\mathbb{P}_+}(E)$ is equal to $\{(C, E), (D, E)\}$ for $\texttt{op}_3$. Thus this instruction can be moved upwards to C and D if predicated by `p8` and `p9` there, respectively. B and A,

[4]It would be easily possible to model that these initializations are automatically inserted in the schedule if and only if they are needed. The next section outlines the adaptations that would allow this. However, the number of needed initializations is often zero or low as experienced in Chapter 7, so that it is viable to insert them in a free slot afterwards manually.

however, are marked twice and thus no candidate blocks—$\mathtt{op}_3$ could only be scheduled there if guarded by `p8` $\vee$ `p9`, but this value cannot be expected to be available in a predicate register.

Extending the candidate block ranges of instructions via predication requires some postprocessing after the ILP solver has delivered the schedule. If a copy of such an instruction is scheduled in a speculative destination block, then the controlling predicate—or the predicate register the block is marked with—must be inserted into the instruction's opcode as qualifying predicate.

It is possible and advisable to compute these associated predicate registers not only for non-speculative, but also for speculative instructions: If such a register is available in the final schedule at the point of a speculatively scheduled copy—which is not guaranteed as these instructions are not made dependent on the controlling compares—then it can also be inserted as qualifying predicate, eliminating the speculativeness of the copy at no additional cost. This has the potential benefit that it prevents speculative stalls as described in Sec. 2.2.4 since predicated-off instructions cause no (or only reduced) stalls (see Sec. 2.2.3.3).

If the controlling predicate is *not* available, but instead another one that is a logical implication of it, then the latter can also be taken as a qualifying predicate in order to *reduce* (instead of eliminating) the speculativeness of the scheduled copy. The benefits of this procedure, known as *predicate promotion* [MLC$^+$92], are the same as described above.

In Fig. 6.1, for example, if a control speculative load is moved from block E to block A, it can be guarded by `p1` there (if available). There is no disadvantage that could result from this, but only potential advantages. In the case of a control speculative load, these include also a reduction of adverse cache effects that possibly result from speculative execution.

As noted above, a controlling compare must not be speculated itself because it controls the execution of non-speculative instructions. However, it can itself be scheduled speculatively via predication again. As an example, the following instruction sequence could be placed in block A of Fig. 6.1 in order to schedule $\mathtt{op}_1$ there:

```
       p1,p2=cmp ;;
(p1)   p5,p6=cmp ;;
(p6)   op1
```

Interestingly, the opposite predefinition would also be possible, namely to speculate compares and the instructions they control *not*. In the above case then both compares could be scheduled in block A without qualifying predicate and *in parallel*, but $\mathtt{op}_1$ could not be moved upwards across block B. It can be expected that dependent instructions rarely benefit from an early speculative execution of a controlling compare since they themselves are then forced to be scheduled non-speculatively with respect to the compare's source block.

However, exceptions are imaginable, and it would be possible to cover these cases by means of a formulation that dynamically switches between both possibilities. We do not expand on this extension here which would further increase the complexity of the ILP model. Instead we opt for the first, more promising predefinition and regard compares as non-speculative instructions. Further arguments in support of this will emerge in Sec. 6.1.2.

6.1.1 Procedure Calls

In contrast to all other branches, a procedure call is considered as a normal non-speculative instruction in the set V (with some special properties, however). In the context of scheduling, this instruction is interpreted as if it would *itself* execute the whole procedure; in other words, it summarizes the data flow to and from the procedure. This leads to a complex input/output pattern determined by the calling conventions of IA-64 [Int01a]:

The instruction reads the output area of the register stack frame, namely the general registers from number $32 + sol$ to number $32 + sof - 1$, and may modify all those registers and all static registers, except for `r4`-`r7` and (usually) the stack pointer in `r12`, which are *preserved*. All predicate registers except for `p6`-`p15` are also preserved. It reads and writes memory like several loads and stores combined, in line with interprocedural aliasing information [Muc97]. Data dependence edges to and from the call are added according to this data flow.

The interprocedural control flow related to a call is not recorded in the CFG, thus calls do not end basic blocks as expressed in Def. 3.2.2. They can be moved within a basic block and even globally between them. The dispersal rules, however, enforce that they are always located after all non-branch instructions in an instruction group (see the end of Sec. 2.2.3.2). To avoid that other instructions are scheduled in the same group that impede this, all outgoing dependence edges of a call with latency zero are turned into edges with latency one. In addition, the latency of all incoming edges is set to zero—this can be done because in effect, these edges do not model dependences related to the call instruction itself, but to other instructions in the called procedure. As a result, instructions that write the parameters to be passed to the procedure can be scheduled in the same instruction group before the call, as shown in the following bundle:

```
{ .mmb
      sub r73=r3,r21           // (output area r72-r74)
(p14) add r72=r69,r22
      br.call.sptk.many b0=flush_block# ;;
}
```

After the return from the procedure, execution continues at the next instruction group after the call in the basic block. This is why calls should never be scheduled into the last instruction group of a block where conditional branches might be located *after* the call—otherwise they would not be executed after the return. To prevent calls from being scheduled into the last group, it is sufficient to change the condition of the first sigma sign of their block length constraints (5.3.14) from $0 \leq t_n < t$ to $0 \leq t_n \leq t$.

Finally, to make sure that not more than five instructions are put together with a call into an instruction group, a call type C can be added as a further predecessor of D in the resource flow network of Fig. 5.11. The network inflow resource constraints generated for the dispersal window type D then take calls into account. However, it is not necessary to add resource constraints for C itself: False dependences between successive calls avoid that more than one of them is scheduled per cycle.

6.1.2 Conditional Branches

Conditional branches are special instructions in the sense that they are always located in the last cycle of a basic block, a position that depends on the block's schedule length. But not only the location of branches is variable, also the number of branches that have to be scheduled there varies with the number of successor blocks in the schedule, as analyzed below. Thus this number cannot be determined *a priori*, yet we may assume in advance that not more than d_A branches are needed in a block A (the constant will be defined later).

Because of their special nature, conditional branch instructions are not included in the set V, but are inserted into instruction groups after scheduling and before bundling. It is important that the exact number of needed branch instructions is taken into account by the resource constraints generated for the dispersal window type $\mathsf{D} \in \mathcal{I}$—otherwise possibly not enough unused execution slots would be left for them in the last instruction group. For this purpose, we create new x variables for branch instructions, namely for each $A \in \mathcal{B}$ and for all $t = 0, \ldots, \mathbb{G}_A$ the variables $x^{At}_{br_1}, \ldots, x^{At}_{br_{d_A}}$. The variables of the respective cycles are then added to the left-hand side of the resource constraints (5.3.13) generated for the type D.

Several constraints are necessary to define these variables, and along with this the correct location and number of branches. The following inequalities make sure that branches are inserted in no other cycle than the last one of a block (which is the only cycle t with $B^A_t = 1$):

$$x^{At}_{br_j} \leq B^A_t \qquad \forall A \in \mathcal{B},\, \forall t \in \{0, \ldots, \mathbb{G}_A\},\, \forall j \in \{1, \ldots, d_A\}$$

The placement of exactly X branches in a block can be enforced by these constraints:

$$\sum_{t=0}^{\mathbb{G}_A} \sum_{j=1}^{d_A} x^{At}_{br_j} = X \qquad \forall A \in \mathcal{B} \tag{6.1.1}$$

It remains to determine a term for the placeholder X, which is the more complex part. This is because the ILP solver may empty blocks completely by moving all instructions out of them, with consequences for the branch structure: Branches to these collapsed blocks disappear and possibly have to be replaced by branches to (not collapsed) successors of them. This did not happen rarely during our experiments, where almost 10% of all blocks were collapsed (see Sec. 7.3.2). Hence a precise model of the branch structure is needed.

We first examine which blocks *can* be emptied: A block A is said to be *collapsible* if all instructions can be scheduled elsewhere, that is, if $\forall n \in V : \Theta^x(n) \setminus \{A\} \neq \emptyset$. Let $\mathcal{B}^\nabla \subseteq \mathcal{B}$ be the subset of all collapsible blocks. A block $A \in \mathcal{B}^\nabla$ is actually *collapsed* in a schedule if and only if $B^A_0 = 1$. Then it must also contain no branches—this is why we have created the variables x^{At}_{br} above not only for $t = 1, \ldots, \mathbb{G}_A$, but also for the nonexisting cycle $t = 0$: These dummy variables $x^{A0}_{br_1}, \ldots, x^{A0}_{br_{d_A}}$ represent no scheduled branches, they just ensure that Equ. (6.1.1) is satisfied even if $B^A_0 = 1$.

Now we analyze which branches can possibly be *taken* in the schedule: When leaving a basic block A, the control flows directly to one of those successor blocks to which a path in G_B exists on which all inner blocks (if existing) are collapsed—these blocks are termed *scheduled* *direct successors* as they are determined by collapsed blocks, which are determined by the schedule. Branches to all these blocks have to be included in the last instruction group of A—except possibly for one that can be placed directly after A in memory so that it can be reached via a fall-through edge. Hence the order of basic blocks in memory also affects the number of needed branches and should be modeled in the ILP, too.

In order to determine the set of scheduled direct successors within the model, we first form the set of *potential* scheduled direct successors, $SC(A) \subseteq \mathcal{B}^{\succ}(A)$: it contains all basic blocks that can be reached from A via a path on which all inner blocks are collapsible. We introduce for all $B \in SC(A)$ binary variables with the following semantics:

$sc_B^A = 1$	$\Leftrightarrow$	B is a scheduled successor of A.
$ft_B^A = 1$	$\Leftrightarrow$	B is placed directly after A in memory so that it can be reached from there via a fall-through edge.

Obviously, a fall-through edge (A, B) can only exist if B is a scheduled direct successor of A:

$$ft_B^A \le sc_B^A \qquad \forall A \in \mathcal{B}, \forall B \in SC(A)$$

Furthermore, fall-through edges are naturally limited by the fact that only one block can be placed *before* or *after* another block in memory:

$$\sum_{B \in SC(A)} ft_B^A \le 1 \qquad \forall A \in \mathcal{B} \qquad\qquad \sum_{A: B \in SC(A)} ft_B^A \le 1 \qquad \forall B \in \mathcal{B}$$

Returns are regarded as branches to the block Ω, which is defined as non-collapsible and to which no fall-through edge is possible.

On the basis of these variables, the number of necessary branches in a block A is determined by the following expression which is substituted into the right-hand side of Equ. (6.1.1):

$$X := \sum_{B \in SC(A)} sc_B^A - \sum_{B \in SC(A)} ft_B^A$$

It remains to be shown how the values of the sc variables can be extracted from the schedule. We recall that sc_B^A must be equal to one if there exists a path from A to B on which all inner blocks (if existing) are collapsed, but not B itself.

If B is a direct successor of A in G_B, then the variable sc_B^A can be directly replaced by one (if $B \notin \mathcal{B}^{\nabla}$) or $\left(1 - B_0^B\right)$ (if $B \in \mathcal{B}^{\nabla}$). Otherwise, if there is a block $C \in SC(A)$ that is *not* a direct successor of A, then we can determine sc_C^A recursively by using those direct successors B of A such that $C \in SC(B)$ (of which at least one must exist after the definition of SC): There exists a path from A to C with collapsed inner blocks if there exists such a path from B to C *and* B is collapsed itself:

$$sc_C^B = 1 \wedge B_0^B = 1 \Rightarrow sc_C^A = 1$$

Or, expressed as a linear inequality:

$$sc_C^B + B_0^B - 1 \leq sc_C^A \qquad \forall A \in \mathcal{B}, \forall C \in SC(A), \forall (A,B) \in E_B : C \in SC(B) \tag{6.1.2}$$

The cardinality of $SC(A)$ is a trivial upper bound on X, the number of needed branches in a block. The bound can be tightened by observing that not all blocks in this set can be scheduled direct successors at the same time: for instance, we can ignore during the counting all blocks that dominate other blocks in $G_B\left[SC(A) \cup \{A\}\right]$. The constant d_A is set to this bound, but never greater than three, the maximum number of branches in an instruction group. If many blocks are collapsed, this limitation can lead to the emergence of single-cycle "trampoline blocks" that just contain branches to other blocks.

To demonstrate the functioning of the formulation, we assume that the blocks B, C, D, E, and G in Fig. 6.1 are collapsible. If in a schedule only the blocks B and D are collapsed, then $sc_E^D = 1$, $sc_C^B = 1$ (since E and C are not collapsed), and $sc_F^D = 1$ (since F is not collapsible). These values are propagated upwards to block A via the inequalities (6.1.2). As a result, $sc_C^A = sc_E^A = sc_F^A = sc_G^A = 1$, so that—if we suppose a fall-through edge to C—three conditional branches to E, F, and G with the qualifying predicates `p9`, `p10`, and `p2`, respectively, must be scheduled there.

If the blocks B, C, and D are collapsed, then a branch from A to E becomes necessary (if $ft_B^A = 0$) that should be taken if `p8` $\vee$ `p9`—a boolean value that is possibly not available in a predicate register. This shows a limitation of the current formulation: it makes sure that the correct *number* of branch instructions can be inserted at correct positions in the schedule, but it does not guarantee that their conditions are computed in predicate registers if the former are disjunctions of predicates. Such cases are rare and we currently remedy them by inserting additional compares into free slots manually. Nevertheless, a future extension could provide a more sophisticated management of predicates that automatically schedules compares that compute conditions as they are needed for branches and code motion.

To make sure that conditions consisting of a single predicate are always ready, we forbid downward motion of compares across the BBG edges they control (that is, whose associated predicate register they generate). It is also important that compares are considered non-speculative, as it has already been predefined in Sec. 6.1.

If in Fig. 6.1 the blocks C, D, E, and G are collapsed, then the whole structure can be merged into a branchless concatenation of the remaining blocks A, B, and F, connected by fall-through edges. This demonstrates the flexibility of the formulation, especially if used in combination with predication: Blocks can be emptied by means of predicated code motion (as previously described) so that the encompassing branch structure disappears. In other words, the decision to apply if-conversion is integrated into the model.

It would be easily possible to incorporate also branch misprediction penalties into this decision: If an estimate is available that a branch from block A to B would have a misprediction probability of ε_B^A, then the term $6 f_A \varepsilon_B^A sc_B^A$ added to the objective function would take the six-cycle penalty of a misprediction into account (see Sec. 2.2.3.1).

6.2 Speculation

6.2.1 Control Speculation

Control speculation in the broader sense is already included in the model via speculative destination blocks. However, in the narrower sense the term means use of control speculative loads. In this section, we focus on how the decision to use them can be incorporated into the model. After that, the next section shows that the formulation can be easily extended towards advanced (data speculative) loads.

Sec. 2.1.5.1 has already made a distinction between speculative and non-speculative instructions. Basically, there are two reasons why an instruction cannot be speculated:

- it could falsely trigger an exception, or
- it could falsely overwrite a live value.

The first point applies especially to memory instructions. As described in Sec. 2.1.5.1, the Itanium architecture has control speculative loads to overcome this restriction for loads. The second point can in principle affect any instruction that writes a destination register that is shared by another definition. It can often be avoided by *renaming* the destination register to a different, previously unused register (together with the source registers of the reached uses).

Thus we rename the destination registers of all definitions—or of as many as *possible*—to different registers prior to scheduling. We can introduce new virtual registers for this purpose. This facilitates not only speculation, but it also removes as many false dependences as possible, increasing the scheduling freedom. In doing so, we decide on the tradeoff between scheduling and register allocation in favor of scheduling (see Sec. 3.1). Theoretically, this can lead to schedules for which no register allocation exists afterwards (if not all virtual registers can be mapped back to architected registers). But as confirmed by the experiments, this is unlikely on the Itanium with its 128 architected registers.

However, as indicated above, we cannot expect that after the renaming *all* instructions have different destination registers, like in a single static assignment form. Instead, we require the *maximal renaming premise*, which says that two definitions may write the same destination register after the renaming if and only if they are *concurrent*:

Definition 6.2.1 (Reaching/Concurrent Definitions (cf. Sec. 1.3.2)) The definitions that reach a given use are termed its *reaching definitions*. If the set of reaching definitions contains two or more instructions, then these instructions are termed *concurrent definitions*. A use that is reached by a definition is called *exclusive use* if it is reached only by this definition, and *non-exclusive* otherwise. □

During the computation of reaching definitions, we must also take those values into account that flow from outside into the scheduling region. To represent this data flow, we insert a special instruction α_A at the beginning of each entry block $A \in \mathcal{B}_{entry}$ that is defined to write exactly these values. Similarly, the last instruction ω_A of each exit block $A \in \mathcal{B}_{entry}$ is an artificial use that reads the return values of the scheduling region (if any). These instructions are not scheduled, they just ensure the correct identification of concurrent definitions.

A concurrent definition can reach multiple exclusive uses, but by definition it must also reach at least one non-exclusive use. During program execution, it depends on the control flow *which* of several concurrent definitions actually reaches a non-exclusive use. Hence, if we execute one of the definitions speculatively, it could falsely overwrite the value written by this definition that would reach the use in the original program. Thus concurrent definitions must be considered non-speculative.

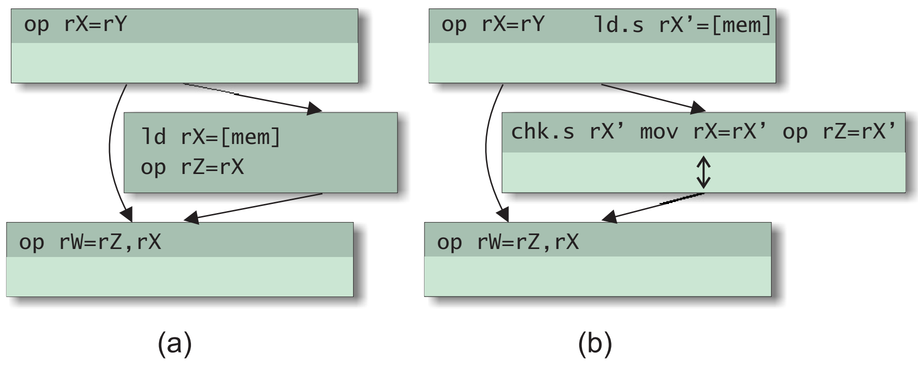


Figure 6.2: Control speculation example with candidate "`ld rX=[mem]`". The resulting schedule length reduction is depicted by the double-headed arrow.

Figure 6.2 (a) shows an example of this: the load could be moved to the uppermost block using control speculation, but the result register `rX` must not be written speculatively because the load is concurrent with "`op rX=rY`". If this is disregarded, the non-exclusive use "`op rW=rZ,rX`" could read the wrong value.

The right-hand side (b) directly shows a possible solution: We can let the load write to a new temporary register `rX'`. *Exclusive* uses like the "`op rZ=rX`" can directly read this register and can possibly also be speculated with the load, whereas for the "`op rW=rZ,rX`" we insert a `mov` instruction that moves the value back to the original register. All *non-exclusive* uses are dependent on this new `mov` instruction, which must—like the `chk.s`—be treated as a non-speculative instruction.

The described use of speculation can be regarded as a general *scheme* that replaces a non-speculative instruction—also called the *candidate* in the following—by its speculative version plus auxiliary instructions (in the above instance, the candidate is "`ld rX=[mem]`", the speculative version is the "`ld.s rX'=[mem]`" and the auxiliary instructions are the `mov` and the `chk.s`). This scheme is flexible in the sense that it is also possible to speculate a concurrent definition that is not a load and vice versa—the `chk.s` and the `mov` are then dropped, respectively. This means that the candidate and its speculative version do not have to be loads, as they are above. If a predicated candidate is to be speculated, then this can be done by omitting the predicate register in the speculative version—but it must then be used to guard both the `chk.s` and the `mov`.

The scheme is employed in two stages: During the generation of the ILP *possibilities* to use this kind of speculation are detected in advance and integrated into the model. The ILP solver then decides whether to make use of them.

As we will detail below, the data dependences of the candidate and its speculative version differ so much that they can be regarded as different instructions. Thus we include them as two separate instructions and model that one of both should appear in the schedule. More precisely, one of two *mutually exclusive sets of instructions* should appear: The first consists of the candidate (a load and/or a concurrent definition), denoted by $n \in V$ here, and the second of its speculative version plus the `chk.s` and/or the `mov`. In the following, the first and the second set are denoted by $\overline{\Delta_n}$ and Δ_n^C, respectively.

To realize that either the instructions from $\overline{\Delta_n}$ or those from Δ_n^C appear in the final schedule, we define a new binary variable S_n as a "speculation switch". Then we replace the right-hand side of the assignment constraints (5.3.10) by $(1 - S_n)$ and S_n for instructions from the first and second set, respectively. Consequently, if one of the sets $\overline{\Delta_n}$ and Δ_n^C is "switched off" via S_n, then the modified assignment constraints enforce that the x and a variables of the contained instructions are all equal to zero.

As intended, all other constraints of the ILP are not affected by instructions "switched off" in this way—except for those instances of the global precedence constraints (3.3.5) where m is such an instruction: Then $(1 - a_m^{\uparrow B})$ is equal to one, which forces all other terms on the left-hand side of the inequality to be zero. To avert this unwanted effect, we add S_n and $(1 - S_n)$ to the right-hand side of these instances if m is element of $\overline{\Delta_n}$ and Δ_n^C, respectively, in order to *relax* them.

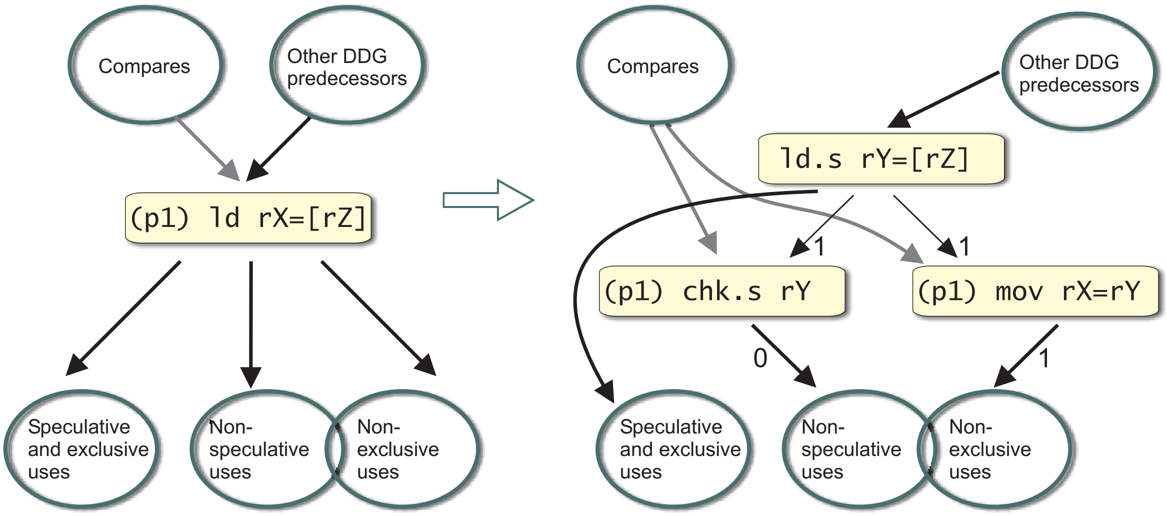


Figure 6.3: Data dependences in the sets $\overline{\Delta_n}$ (left) and Δ_n^C (right).

The `chk.s` and/or the `mov` are added as new, separate instructions with the same source block as the candidate and with their own data dependences. Figure 6.3 depicts how these dependences are created for the most complex case, namely for a predicated candidate that is both a load and a concurrent definition. The left-hand side shows different sets of true data dependences related to this load as arrows. It is distinguished between three groups of dependent instructions: both speculative and exclusive, non-speculative, and non-exclusive uses, the latter two of which may overlap, as illustrated in the figure. The right-hand side shows how these dependences are split up when the speculation scheme is used: the first group can be speculated together with the `ld.s`, but the second and the third group are dependent on the check and the move (with latency zero and one[5]), respectively.

If both the candidate and one of its uses are loads, and if the former loads the address operand of the use, then the total latency between both instructions is two (according to Sec. 2.2.3.3). Then an additional true dependence edge with this latency has to be added between the speculative version and this use in order to allow for the increased latency (not shown in the figure). Otherwise these two instructions could be scheduled only one cycle apart.

It is important that these speculation possibilities (the new instructions and dependences) are added in reverse topological order of the DDG. In doing so, newly added speculative versions of instructions can act as "speculative and exclusive uses" itself when the speculative versions of DDG predecessor instructions are added. This allows to *cascade* several dependent speculative versions, i.e., to schedule a whole sequence of these instructions speculatively.[6]

Furthermore, a speculation possibility for a candidate should only be added if a benefit can be expected from using it. This can be excluded in one of the following cases:

- The candidate does not have an exclusive use. Then all uses are dependent on the `mov`, which does not allow to schedule them earlier. Candidates that are floating-point loads constitute an exception since their latency is longer than that of a floating-point move (6 vs. 4 cycles).

- The source block of the candidate does not have predecessor blocks where a speculative version could be scheduled speculatively (that is, blocks that are not postdominated by the source block). For example, it does not make sense to speculate loads in the entry block—the speculative version could not be scheduled earlier than the candidate itself.
  However, this rule does not hold for a predicated candidate—there the speculative version can in principle be scheduled earlier in the same block since it is not dependent on the compare(s) that generate the predicate. Remarkably, if there is a predicated candidate that is not a load and has no non-exclusive use, then the qualifying predicate is unnecessary and can directly be removed—then there is no need to postpone this decision to the ILP solver.

If the candidate is unpredicated, then its source block can even be excluded from the candidate block set of the speculative version in order to decrease the ILP size. This is a consequence of the last point. As a result, this instruction has its own source block not included in its candidate

[5]The scheme can also be applied to floating-point instructions; then the latency of `mov fX=fY`, which is a pseudo-op for `fmerge.s fX=fY,fY`, is four cycles.

[6]In Sec. 2.1.5.1 we have noted that checks of the result(s) computed by this sequence only are already sufficient to detect all possible exceptions. Our current model does not support this simplification and could introduce more checks than necessary. This was not addressed so far as it was not relevant in practice.

block range. This may sound problematic at first, but we have never required that the source block is also a candidate block (although it is normally the case).

In the final schedule, recovery code must be added to which the checks branch if there are deferred exceptions. In practice, such a branch is a very rare event that happens in less than 0.001% of all cases [Alp03]. Nevertheless, individual speculative loads could experience higher failure probabilities or incur rare but significant penalties if they miss the L1 cache or the TLBs [Int04]. Thus, the use of control speculation is ideally guided by a cost model that estimates these effects. Such information—if available from static analysis, heuristics or profiling—can be integrated into the objective function of the model to increase its precision.

A cost model was not available during our experiments in Chapter 7, but we have excluded there—as a general measure—code motion of speculative loads into blocks whose execution frequency is by a factor k times higher than that of the source block, i.e., we forbid control speculation that is likely to be useless (we used $k = 5$ in the experiments).

6.2.2 Non-Exclusive Use Forking

The speculation scheme presented in the previous section can, as remarked there, only then be profitably applied if the candidate has an exclusive use. This is because all non-exclusive uses must respect the dependence on the new non-speculative move to ensure that they read the result of the candidate only if the speculation succeeds (and otherwise that of another, concurrent definition). Fig. 6.4 shows such a case where speculating the concurrent definition "`add r8=-16,r10`" (in (b)) would not allow to schedule the non-exclusive use "`sub r11=r11,r8`" earlier than before (in (a)).

However, we *can* schedule the use earlier if we split it up into two versions: one which reads the result of the speculative version of the use, "`add r8b=-16,r10`", and another which reads `r8` as written by the other concurrent definition, "`ld4 r8=[r33]`". As depicted in Fig. 6.4 (c), these two speculative versions of the non-exclusive use are designed to write their results to two new temporary registers `r11a` and `r11b`, respectively. Once the predicates are available in cycle three, the correct result is written back to `r11`.

We can go even further and fork not only non-exclusive uses, but also other instructions that depend on them. In doing so, we effectively execute two versions of the *same* sequence of instructions in parallel, one for each of the two possible outcomes of a (yet to be made) control flow decision (which is represented by the predicates in the example). They are executed concurrently and once the decision is resolved, the results of the correct sequence are moved back to the original registers and those of the other one are discarded.

In the example, we have not only forked the use, but also the dependent store. However, this instruction is special in the sense that it is non-speculative. Such instructions can also be forked, yet not speculatively; instead the two versions must be guarded by the predicates that represent the disjoint control flow paths. In this regard, we distinguish between *speculative* and *predicated forking*. Predicated forking can save one cycle if applied to the last instruction in a forked chain since it avoids the data dependence(s) on the move(s). In Fig. 6.4 (c), for instance, without forking the store "`st4 [r33]=r11`" had to be scheduled in cycle four in order to respect the RAW dependences on the last two moves in cycle three.

Cycle			

1	`ld4 r9=[r32]`
	`ld4 r8=[r33]`
		`ld4 r9=[r32]`
		`ld4 r8=[r33]`
		`add r8b=-16,r10`
			`ld4 r9=[r32]`
			`ld4 r8=[r33]`
			`add r8b=-16,r10`

2	`cmp.eq p1,p2=0,r9`	`cmp.eq p1,p2=0,r9`	`sub r11a=r11,r8`
			`sub r11b=r11,r8b`
			`cmp.eq p1,p2=0,r9`

3	`(p1) add r8=-16,r10`	`(p1) mov r8=r8b`	`(p1) st4 [r33]=r11b`
			`(p2) st4 [r33]=r11a`
			`(p1) mov r8=r8b`
			`(p1) mov r11=r11b`
			`(p2) mov r11=r11a`

4	`sub r11=r11,r8`	`sub r11=r11,r8`	
5	`st4 [r33]=r11`	`st4 [r33]=r11`	
	(a)	(b)	(c)

Figure 6.4: Forking and speculating a non-exclusive use.

The critical path length reductions achievable through forking can be drastic, as demonstrated by the 40% improvement in the example. However, the example also shows that it has the potential to triple the instruction count: It duplicates each forked instruction and adds for each a move. At least, this move can be omitted if there exists no other use of the register that is not forked, so that the eventual increase lies somewhere between a doubling and a tripling. Such a trading of an increased number of parallel, speculative instructions for a decreased critical path length complies with the basic principles of the architecture.

Non-exclusive use forking can be incorporated into the model very similarly to the previous speculation decisions using mutually exclusive sets of instructions. We do not expand on the details here, but we make use of a tentative implementation in the experiments (see Sec. 7.2).

The proposed transformation is similar to *multipath execution*, a hardware technique that forks on hard-to-predict branches and executes both resulting control flow paths in parallel until the branch is resolved [ASMC98]. To some extent, speculative upward code motion already implements such a control speculative execution of instructions from two different paths, but the presented transformation goes beyond that in the regard that it duplicates instructions from the point where the two paths join again and speculates them, too. It is an example of how aggressive speculation in combination with predication can shrink the critical path considerably, which is a key requirement for more instruction-level parallelism.

6.2.3 Data Speculation

An inevitable prerequisite for the use of data speculation is the availability of a measure of the likelihood of aliasing: Ideally, we have for each memory dependence edge $e \in E_D^{mem}$ (or at least for the RAW edges) an aliasing probability $\kappa_e \in]0, 1]$ given. Dependences with $\kappa_e = 1$ are called *must-dependences* and the others *may-dependences*. Analogously, we use the terms *must/may-definitions* and *must/may-uses* in the context of such dependences. In the construction of the global data dependence graph in Def. 3.2.6, it is important that the exclusion criterion (the second point) applies only to must-definitions and must-uses.

Control Speculation				Data Speculation			
Loads		Checks		Loads		Checks	
Variant	Mnmc.	Variant	Mnmc.	Variant	Mnmc.	Variant	Mnmc.
Control speculative load	`ld.s`	(Control) speculation check	`chk.s`	Advanced load	`ld.a`	Check load	`ld.c`
				Speculative advanced load	`ld.sa`	Advanced load check	`chk.a`

Table 6.1: Overview of the speculation-related instructions from Sec. 2.1.5. The notions from [Int02a] are given together with the mnemonics.
(Note: the table does not imply a correspondence between loads and checks in the same line.)

We can employ data speculation separately or in combination with control speculation (using the advanced loads `ld.a` and `ld.sa`, respectively; a short overview of the notions is provided by Table 6.1). In both cases, only a check load `ld.c` or an advanced load check `chk.a` is required instead of the `chk.s`. The resulting main difference as regards the scheduler is that the former two instructions can only be executed on ML units, while the control speculation check can issue to MS or I. This difference is significant in practice since the two ML units can be considered as scarce resources. Moreover, it forces us to consider data speculation checks as *separate* instructions—they cannot be merged with the control speculation check of the speculation scheme from the previous section, at least not without complicated changes to the resource constraints.

At least, we can regard the `ld.c` and the `chk.a` as one single instruction in the ILP. We refer to it as the "combined check" `ld.c/chk.a` in the following. After scheduling, this instruction is turned into an `ld.c` or into a `chk.a`, as described below. Figure 6.5 depicts how it is incorporated into the speculation scheme (initially, all dotted arrows can be ignored): All potentially aliasing stores[7] are separated from the group "Other DDG predecessors" and form a group of their own. Let $ST = \{st_1, \ldots, st_k\} \subseteq V$ denote the set of these stores. The speculative version

[7]In principle, a store is "potentially aliasing" with a subsequent load if for the memory dependence edge e between the two holds $\kappa_e < 1$. In practice, however, memory dependences with $\kappa_e > 0.1$ can already be regarded

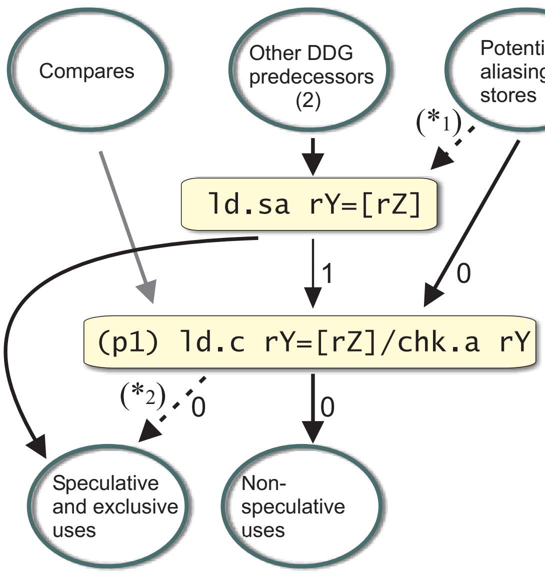

Compares
Other DDG predecessors (2)
Potentially aliasing stores
(*1)
ld.sa rY=[rZ]
1
0
(p1) ld.c rY=[rZ]/chk.a rY
(*2)
0
0
Speculative and exclusive uses
Non-speculative uses

Figure 6.5: Data dependences in Δ_n^D .

is no longer dependent on them, but the combined check instruction is. All speculative and exclusive uses are dependent on the speculative version, but all non-speculative uses are dependent on the combined check with latency zero—this is the same as with the control speculation scheme of Fig. 6.3.

If only the use of data speculation should be made possible, then this scheme can be used in a stand-alone way: the "`ld.sa rY=[rZ]`" is replaced by an advanced load "`ld.a rY=[rZ]`", which is a *non-speculative* instruction. Consequently, it must be guarded by the same predicate register as the candidate and made dependent on the same compares (differing from Fig. 6.5).

Alternatively—and as originally depicted by the figure—the possibility to use control *and/or* data speculation can be included in the ILP. Fig. 6.5 can then be considered as an addendum to the scheme of Fig. 6.3. Both are to be combined in such a way that "`ld.s rY=[rZ]`" and "`ld.sa rY=[rZ]`" are the same instruction. Altogether, three mutually exclusive (but partly overlapping) sets of instructions are distinguished:

- $\overline{\Delta_n}$: The candidate.
- Δ_n^C: The speculative version plus the `chk.s` and/or the `mov`.
- Δ_n^D: The speculative version plus the combined check `ld.c/chk.a` and/or the `mov`.

(In case of stand-alone control and data speculation, the third and the second set do not exist, respectively).

as must-dependences since the benefit from ignoring them with data speculation will hardly outweigh the penalty cycles due to frequent failures. The choice of this threshold depends on the data speculation failure penalties of the target processor.

Now two question remain to be clarified regarding the integration with the scheduling ILP: How can the decision between control speculation *alone* and combined with data speculation be modeled? How can it be determined whether an advanced load check instruction `chk.a` in place of a simple check load `ld.c` is necessary, and is this distinction important?

The first question can only emerge if the ILP solver should have the choice between either control speculation alone or in combination with data speculation. To model this choice, we employ two new binary variables S_n^C and S_n^D that are intended to be equal to one exactly in the first and the second case, respectively, and add the equation:

$$S_n = S_n^C + S_n^D$$

Then we change the assignment constraints and the precedence constraints in such a way that the sets Δ_n^C and Δ_n^D appear in the schedule if and only if S_n^C and S_n^D are equal to one, respectively. This is done as described in the previous section. Finally, we include instances of the precedence constraints (5.3.12) for the may-dependences marked by $(*_1)$ in Fig. 6.5, but add S_n^D to the right-hand side of them so that these dependences can be ignored if and only if data speculation is being used ($S_n^D = 1$).

Such constraints need *not* to be added for a may-dependence on a store st_i that is a DDG predecessor of another store st_j whose source block postdominates that of st_i. In this case, a violated dependence on st_i already implies a violated dependence on st_j so that the precedence constraints related to st_i are redundant. We denote by $\widetilde{ST} \subseteq ST$ the subset of those stores for which the constraints are *not* redundant.

Regarding the second question, it should be recalled from Sec. 2.1.5.2 that uses can only be speculated together with an advanced load if the advanced load check instruction `chk.a` is used, which branches to recovery code. However, the penalty of failed data speculation is then with 20 cycles and more significantly higher than the 8-cycle penalty if a check load is used (these penalties are denoted by p_{DA} and p_{DC}, respectively, in the following).

There are three ways how to deal with this difference: Firstly, if it can be supposed that a failure is extremely unlikely, then the distinction between the two sorts is dispensable and can be ignored in the ILP. Then *after* the optimization, the combined checks in the schedule are replaced by check loads or, where necessary, by advanced load checks with recovery code.

Secondly, if we have for an advanced load a general estimate of the failure probability that is not negligible (denoted by κ), then this can be taken into account in the objective function. For this we introduce a further binary variable S_n^{DA} that is intended to be equal to one if and only if data speculation is used with an advanced load check instead of a check load. If this variable has value one, then also S_n^D must have value one (since then data speculation is being used), as ensured by the following constraint:

$$S_n^{DA} \leq S_n^D$$

Furthermore, we include precedence constraints for the dependences marked by $(*_2)$ in Fig. 6.5, but this time we add S_n^{DA} to their right-hand sides to model that these dependences can be ignored if and only if an advanced load check is being used ($S_n^{DA} = 1$). Then we can add the term $\left[p_{DC} f_{s(n)} \kappa\right] S_n^D + \left[(p_{DA} - p_{DC}) f_{s(n)} \kappa\right] S_n^{DA}$ to the objective function to take the penalties into account. In this term, the penalty cycles are weighted by the failure probabilities *and* by the execution frequency of the check's source block, $f_{s(n)}$; the terms in the square brackets are constants.

The third and most precise approach breaks the failure probability down into the components contributed to by the different speculated may-dependences. For this purpose, we introduce for each store $st_i \in ST$ a new pair of mutually exclusive binary variables, $S^{DA}_{(st_i,n)}$ and $S^{DC}_{(st_i,n)}$, of which one is equal to one if and only if the advanced load is scheduled before this store, namely the former if $S^{DA}_n = 1$ and the latter if $S^{DA}_n = 0$, respectively. The reason why we need two additional variables is that the objective function must remain linear—otherwise we could have a single variable $S^{D}_{(st_i,n)}$ and use the product $S^{D}_{(st_i,n)} \cdot S^{DA}_n$ instead of $S^{DA}_{(st_i,n)}$, for example. The penalty weighted by the aggregate failure probability—divided into advanced load check and check load parts—is then given by the following sum, which is added to the objective function:

$$\sum_{i=1}^{k} \left(\left[p_{DA} f_{s(n)} \kappa_{(st_i,n)} \right] S^{DA}_{(st_i,n)} + \left[p_{DC} f_{s(n)} \kappa_{(st_i,n)} \right] S^{DC}_{(st_i,n)} \right)$$

Similarly as above, the term $S^{DA}_{(st_i,n)} + S^{DC}_{(st_i,n)}$ is added to right-hand side of all instances of the precedence constraints (5.3.12) that are generated for a may-dependence $(*_1)$ on a store st_i (instead of the variable S^D_n). The following constraints are necessary to connect the S^D_n variable to the new variables. If one of the new variables has value one, then also S^D_n must have value one (since then data speculation is being used):

$$S^{DA}_{(st_i,n)} + S^{DC}_{(st_i,n)} \leq S^D_n \qquad \forall st_i \in \widetilde{ST}$$

Finally, we must add the following inequalities to enforce that the $S^{DC}_{(st_i,n)}$ variables can only be equal to one if no advanced load check is employed:

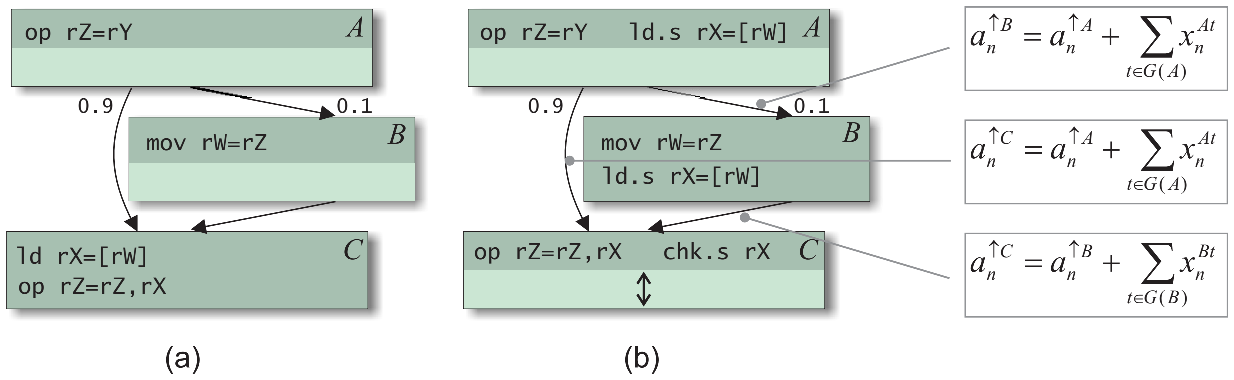
$$S^{DC}_{(st_i,n)} + S^{DA}_n \leq 1 \qquad \forall st_i \in ST$$

After the ILP solver has returned a solution, minor postprocessing is necessary if a data speculation possibility was utilized in the schedule (visible from the value of S^D_n in the solution): Copies of the speculative version are turned into an "`ld.sa`" if they are scheduled speculatively and into an "`ld.a`" else. The combined check is replaced by a check load or an advanced load check depending on whether a use is speculated in the schedule or not. In the latter case, recovery code is also added.

6.3 Partial-Ready Code Motion

In principle, *partial-ready code motion* [BMM00] is a special form of upward code motion in combination with control speculation: it speculates that a particular control flow path is taken and ignores the data dependences from other paths. Fig. 6.6 gives an example where this is profitable.

On the left-hand side (a), we cannot move the load upwards into block A because of the dependence on the `mov`. The load is *data ready* there only under the assumption that the left,



op rZ=rY
A
0.9
0.1
mov rW=rZ
B
ld rX=[rW]
op rZ=rZ,rX
C
op rZ=rY ld.s rX=[rW]
A
0.9
0.1
mov rW=rZ
ld.s rX=[rW]
B
op rZ=rZ,rX chk.s rX
C
$a_n^{\uparrow B} = a_n^{\uparrow A} + \sum_{t \in G(A)} x_n^{At}$
$a_n^{\uparrow C} = a_n^{\uparrow A} + \sum_{t \in G(A)} x_n^{At}$
$a_n^{\uparrow C} = a_n^{\uparrow B} + \sum_{t \in G(B)} x_n^{Bt}$
(a)
(b)

Figure 6.6: Partial-ready code motion.

likely path is taken ("*partial-ready*"). On the right-hand side (b), we ignore the dependence by scheduling the load in block A and insert a compensation copy on the other path that *reexecutes* it after the `mov`, overwriting `rX` with the correct value. An important detail is that a speculative load must be used in A because its load address is undefined if the edge (A, B) is taken. Generally, the idea is to schedule instructions earlier on a path by speculatively ignoring dependences from other paths, and to place compensation copies on the other paths that respect these dependences and, if necessary, overwrite the involved register with the correct value.

Intuitively, we have scheduled the speculative load "around" block B, ignoring all dependences from this block. Note that we could also have done this with a WAW dependent instruction (imagine, for example, that the `mov` was a call instead that would write `rX` as a scratch register), but generally not with WAR dependent instructions. Moreover, dependences $(m, n) \in E_D^{RAW} \cup E_D^{WAW}$ can only then be speculatively ignored if they are not *dominating*, i.e., if $s(m)$ does not dominate $s(n)$. We collect such *ignorable* data dependences in the set $E_D^{PR} \subseteq E_D^{RAW} \cup E_D^{WAW}$; we will later describe how they are chosen. In the context of a dependence $(m, n) \in E_D^{PR}$, we often also say that m and its source block $s(m)$ are ignorable.

We use the following notation to define partial-ready code motion exemplarily for a use n; for the sake of simplicity, it is initially tailored towards true dependences with respect to one source register of n:

Definition 6.3.1 (S-Path, S-Defined) Let a use $n \in V$ and one of its source registers r be given. Let $\{(d_1, n), \ldots, (d_k, n)\} \subseteq E_D^{PR}$ denote the ignorable data dependences with respect to r and $\mathsf{PR}^V(n) := \{d_1, \ldots, d_k\} \subseteq V$ the corresponding ignorable instructions among its reaching definitions. Furthermore, let S_i denote the source block of d_i and $\mathsf{PR}^{\mathcal{B}}(n) := \{S_1, \ldots, S_k\}$ the set of all ignorable blocks. In the context of this use and this source register, we employ the following notation:

1. We refer to a path $C \in \mathcal{C}(S_i) \cap \mathcal{C}(s(n))$ as an S_i*-path* (w.r.t. r) if it traverses none of the other blocks $S_1, \ldots, S_k$ after S_i.

2. A copy of n is called S*-defined* if it is the *latest scheduled copy* of this instruction on an S-path. $\square$

This definition takes into account that partial-ready code motion—abbreviated PR code motion—introduces multiple scheduled copies of the same instruction on a control flow path: Then the *latest executed copy* of this instruction counts since it overwrites all the results of previous copies with the correct value. This can be seen in Fig. 6.6 along the path A-B-C. The latest scheduled copy along an executed path is always *useful* and all copies before are *irrelevant (dead)* (along this path). We define the *speculative scope* of n in a schedule as all those blocks A such that copies of n are scheduled not only in A or its predecessors, but also in some of its successor blocks—then it still depends on the control flow what will actually be the latest executed copy, which writes the value of r definitively.

A copy of n must respect the dependence on a definition d_i in the schedule *if and only if* it is S_i-defined: then and only then there exists an S_i-path on which it is the latest scheduled copy, which must read the value written by d_i. The following definition summarizes these observations:

Definition 6.3.2 (PR Code Motion) A *global schedule with partial-ready code motion* is defined in the same way as in Def. 3.3.3 except for the following relaxations:

1. More than one copy of the same instruction is allowed along each control flow path, i.e., the relation symbol “$=$” in (3.3.2) is changed to “$\geq$”.
2. A copy of an instruction n may ignore the data dependence on a definition $d_i \in \mathsf{PR}^V(n)$ if it is not S_i-defined. □

As an example, Fig. 6.7 shows a scheduling region with three concurrent definitions in the blocks B, C, and F, which constitute—together with the implicit, not shown instruction α_A in A (see Sec. 6.2.1)—the reaching definitions of the use `Y=r1` (which is also denoted by n). Under the assumption that $\mathsf{PR}^V(n)$ is equal to this set of reaching definitions, the right-hand side shows a possible application of PR code motion: The four scheduled copies of `Y=r1` are (from left to right) C-defined, B-defined, A-defined, and F-defined and respect the dependences on the respective definitions in these blocks. The second and the third one are scheduled partial-readily. The latter (the copy in A) is not C-defined, B-defined or F-defined since it is not the latest copy on any of the paths A-B-C-H, A-B-H or A-D-F-G-H. Thus it may—and it does—ignore the dependences on all definitions except α_A.

The speculative scope of the use consists of A, B, and D. Along the C-path A-B-C-H, the first two scheduled copies are executed speculatively and in this case also needlessly as their results are overwritten by the latest scheduled copy (which reads `r1` from `r1=B`). In this example, the transformation *quadruples* the number of scheduled copies of the use—nevertheless, it can be profitable if the path A-D-E-G-H is hot and if this instruction would add to the block length of H if scheduled there, but not in A where it would fit into a free slot.

In principle, we could move all the non-speculative concurrent definitions in Fig. 6.7 also upwards to block A via predicated code motion. Then also the use could be moved upwards to this block, even without partial-ready code motion. However, this would make the definitions—and indirectly also the use—dependent on the controlling compares (not shown in the figure), instructions that are often the last ones in long data dependence chains. When scheduled partial-readily, the use can be executed independently of these chains and thus often earlier.

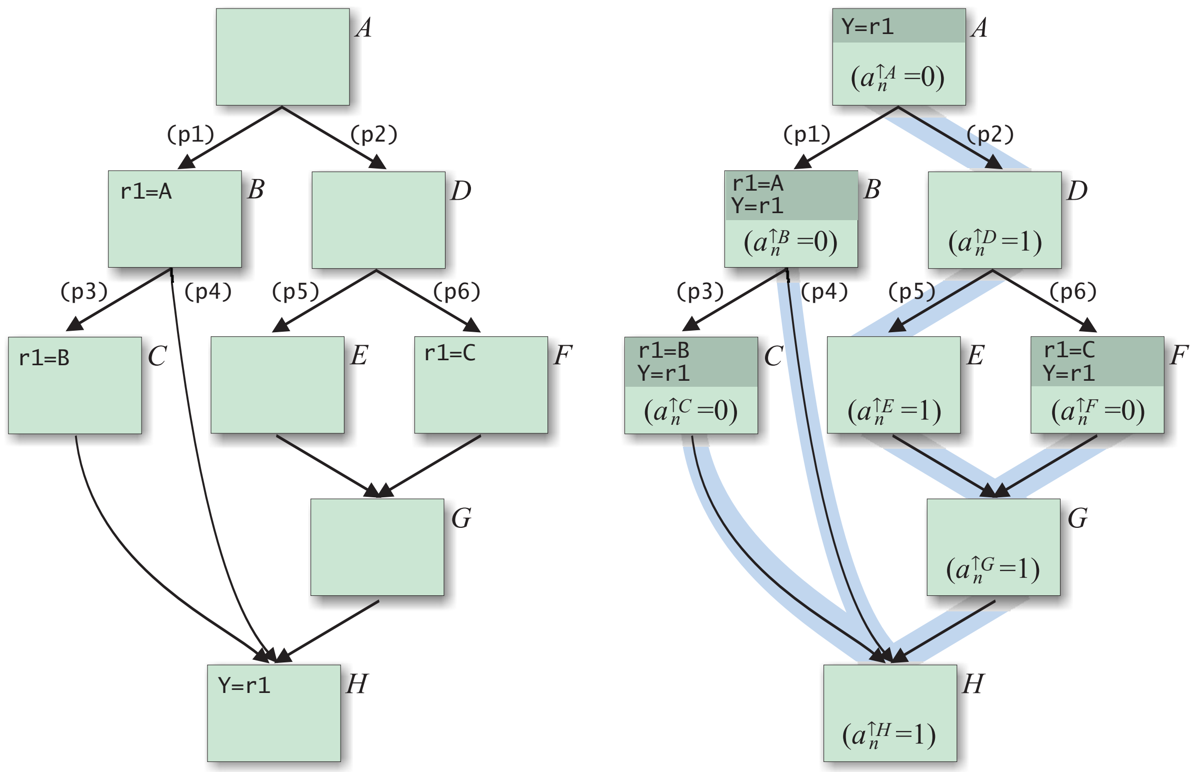


Figure 6.7: Case study of PR code motion.

PR code motion is widely unknown in the literature, but it is employed by wavefront scheduling (see Sec. 3.3.1) where it yields a 20% speedup on individual benchmark programs (the average benefit is approx. 5%) [BMM00]. The support of this transformation in the ILP model requires profound changes since it violates the following two basic assumptions the latter is built on:

Remark 6.3.3 For any global schedule according to Def. 3.3.3 holds:

1. No instruction is scheduled twice on any control flow path (Prop. 3.3.4).
2. If $C \in \mathcal{C}(s(m)) \cap \mathcal{C}(s(n))$ and n depends on m, then any copy of n appears after any copy of m on C (Def. 3.3.3). □

In the example of Fig. 6.7, the use violates both assumptions (regarding the second one, consider the copies of `Y=r1` and `r1=B` scheduled in B and C, respectively). PR code motion is only possible if these assumptions are *relaxed* under certain circumstances, of course without compromising the correctness of the model.

We initially concentrate on the first assumption, especially on how the assignment of the a_n variables in Fig. 6.7 violates the a-x constraints (5.3.11): There the variable $a_n^{\uparrow B}$ has value zero because a copy of this instruction is scheduled in B and the right-hand side of the a-x constraint

(5.3.11) generated for the BBG edge (B, H) cannot grow larger than one. At the same time, the right-hand side of the instance generated for (A, B) is one—but then the left-hand side of this equation, $a_n^{\uparrow B}$, must also be equal to one, yielding a contradiction. One way to resolve this contradiction is to turn instances of equation (5.3.11) into *inequalities* by replacing the “$=$” by “$\leq$”:

$$a_n^{\uparrow B} \leq a_n^{\uparrow A} + \sum_{t \in G(A)} x_n^{At} \qquad \forall n \in V, \forall A, B \in \Theta^a(n) : (A, B) \in E_B \tag{6.3.1}$$

In these *relaxed* a-x constraints, the right-hand side can be one even if the left-hand side is zero. The schedule of Fig. 6.7 is feasible under these modified constraints: In the figure, those BBG edges for which the produced a-x constraints have value one on both sides are provided with a grey shadow. The other edges (A, B), (B, C), and (D, F) are exactly those for which the left-hand side of this constraint has value zero and the right-hand side value one—for those instances the relaxation is needed.

We now investigate which consequences it would have for the correctness of the formulation if *all* a-x constraints were relaxed in this way. Semantically, this step is equivalent to replacing “$\Leftrightarrow$” by “$\Rightarrow$” in the characterization of the a variables on page 103. The relaxed constraints still ensure that *at least one* copy of each instruction is scheduled along every path through its source block (previously: exactly one). However, they threaten to invalidate the following proposition, which is an obvious consequence of the original a-x constraints (cf. Prop. (5.1.24)) and which is important for the data dependence preservation:

Proposition 6.3.4 *In any feasible solution of the global scheduling ILP, a copy of an instruction $n \in V$ is scheduled into a block $A \in \Theta^x(n)$ if and only if $a_n^{\uparrow A} = 0$ and $a_n^{\uparrow B} = 1$ for a direct successor B of A.* □

This proposition can be violated since the right-hand side of Equ. (6.3.1) can take the value two after the relaxation. To make sure that this does not happen, we add the following constraints:

$$a_n^{\uparrow A} + \sum_{t \in G(A)} x_n^{At} \leq 1 \qquad \forall n \in V, \forall A \in \Theta^x(n) \tag{6.3.2}$$

Even with this proposition, data dependence preservation is no longer guaranteed in all cases. The difference is that now there can be multiple scheduled copies of the same instruction along program paths, of which only the last one count. The following corollary characterizes the latter in a solution; it follows from Prop. 6.3.4:

Corollary 6.3.5 *The latest scheduled copy of an $n \in V$ on a path $C \in \mathcal{C}(s(n))$ is placed in block $A \in \mathcal{B}(C)$ if and only if $a_n^{\uparrow A} = 0$ and $a_n^{\uparrow B} = 1$ for all $B \in \mathcal{B}^{\succ}(A) \cap \mathcal{B}(C) \cap \Theta^a(n)$.* □

To analyze the effect of multiple copies on data dependence preservation, we have to take a closer look at what constitutes this preservation: RAW dependences enforce that the two latest scheduled copies of a pair of instructions are scheduled in a certain order so that data can flow between them. False dependences—the WAW and WAR types—then just make sure that no definition is moved from *above* and *below*, respectively, between them that would overwrite the

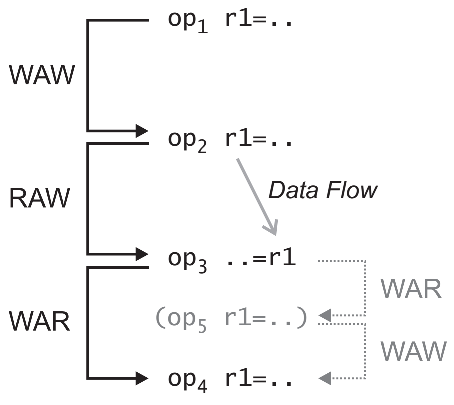


Figure 6.8: Relationships of different data dependence types.

live value and thus interfere with the data flow (see Fig. 6.8). In this way, true dependences model the *requirement* to schedule instructions in a certain range and false dependences the *exclusion*.

Based on this observation, we can *refine* the definition of data dependence preservation—originally of the simple form of Remark 6.3.3-(2)—in such a way that it takes multiple scheduled copies of instructions into account:

Definition 6.3.6 (Data Dependence Preservation) A dependence $(m, n) \in E_D$ is preserved on a path $C \in \mathcal{C}(s(m)) \cap \mathcal{C}(s(n))$ if the copies of both instructions on C fulfill the following requirements, depending on the dependence type:

- $(m, n) \in E_D^{RAW}$: The latest scheduled copy of m is placed before the latest scheduled copy of n.
- $(m, n) \in E_D^{WAW}$: No copy of m is scheduled after the latest scheduled copy of n.
- $(m, n) \in E_D^{WAR}$: No copy of n is scheduled before the latest scheduled copy of m. □

As mentioned before, this more differentiated definition of data dependence preservation is just a refinement of the previous definition with regard to PR code motion—both are equivalent under Remark 6.3.3-(1): then the single scheduled copy on a path is automatically the latest copy, so that Remark 6.3.3-(2) is equivalent to each of the three requirements.

We now examine the impact of the relaxation on dependences of different types. For this purpose, we make use of the following inequality, which is implied by the precedence constraints (5.3.12) (cf. Equ. (5.1.20)):

$$a_n^{\uparrow A} \le a_m^{\uparrow A} \qquad \forall (m, n) \in E_D, \, \forall A \in \Theta^a(m) \cap \Theta^a(n)$$

Corollary 6.3.5 ensures in combination with this inequality and Equ. (5.3.3) from Def. 5.3.1 that the latest scheduled copy of m is always placed in the same block or earlier than the latest scheduled copy of n on any path $C \in \mathcal{C}(s(m)) \cap \mathcal{C}(s(n))$. As a result, the schedule complies with Def. 6.3.6 for RAW and WAW dependences, even if all a-x constraints are relaxed.

WAR dependences, however, are not necessarily preserved because the relaxation enables the placement of a further copy of n before the latest scheduled copy of m. Our current remedy to this is to forbid the relaxation of the a-x constraints (5.3.11) for all instructions that are WAR dependent on others, at least if the constraint is instantiated for a pair (A, B) such that A and B are candidate blocks of these other instructions. Fortunately, WAR register dependences are rare in the DDG since we have removed as many false dependences as possible via register renaming ("maximal renaming premise"). WAR memory dependences are more frequent, but since PR code motion cannot be applied to stores anyway (see below), this is of no consequence. Hence the impact of this restriction is only minor.

The special nature of WAR dependences has also the consequence that we must be careful when removing them: In Def. 3.2.6, a WAR dependence between two instructions is not included in the DDG if there is an intermediate definition of the same value along all paths between the two instructions. As depicted in Fig. 6.8 by the dotted edges, this can be done because then there is a WAW dependence on this intermediate definition (op_5), which is itself WAR dependent on the use (op_3). This renders the WAR dependence edge redundant according to Def. 3.2.7. However, PR code motion could cause the WAW dependence to be violated on some paths—at least if it is not dominating (that is, if the source block of the intermediate definition does not dominate that of the later definition). Thus in these cases the exclusion criterion in Def. 3.2.6 must not be applied and the WAR dependence edge must be included in the DDG.

The same effect must be allowed for during the minimization of the data dependence graph as described in Def. 3.2.7: Only a chain of *dominating* dependences can render a WAR dependence redundant. Moreover, the relaxation is only allowed for instructions that are *multiply executable* without a changing semantics, which is not the case for instructions whose input and output operands overlap, such as `add r1=1,r1`[8] or memory operations with post-increment.

A further prerequisite of PR code motion is an adaptation of the valid candidate block ranges (Def. 5.3.1). There we have excluded the possibility to schedule an instruction before all the candidate blocks of another instruction it depends on Equ. (5.3.2). The basis of this removal is Rem. 6.3.3-(2), which may be violated by partial-ready copies. If we assume, for example, that in Fig. 6.7 the candidate blocks of the concurrent definitions comprise only their respective source blocks, then according to Equ. (5.3.2) the blocks A, B, and D would be excluded from the candidate block set of the use, making PR code motion impossible.

Thus Equ. (5.3.2) must in this general form not be applied to the ignorable DDG edges. It can be omitted completely since the dependence preservation does not rely on it (see the discussion of data dependence preservation after Def. 6.3.6). However, Theorem 5.3.3 is based on it and must no longer be applied under these circumstances.

Instead of omitting (5.3.2), it is recommended to apply a modified variant, adapted to the edges in E_D^{PR}, to keep the candidate block ranges as small as possible. For this, we exploit that these dependences must be preserved at least in certain blocks: If n is scheduled in a block A that is *postdominated* by the source block of a definition $d_i \in \mathsf{PR}^V(n)$, then it must respect the

[8]It may be possible to prevent via renaming that a register occurs at the same time as a source and destination register. However, this is not possible if the instruction is RAW dependent on another instruction that is at the same time a concurrent definition with respect to this register. This can especially occur if the instruction is inside a loop (see Sec. 6.4).

dependence on this instruction; thus, if A is located before all candidate blocks of d_i, then it cannot be a candidate block of n:

$$s(d_i) \text{ pdom } A \wedge A \in \mathcal{B}^{\frown}(\Theta^x(d_i)) \Rightarrow A \notin \Theta^x(n) \tag{6.3.3}$$

Similarly, if for a block A a subset $D \subseteq \mathsf{PR}^V(n)$ exists such that every program path through A passes also through at least one of the source blocks of the definitions in D, then a copy of n scheduled in A must respect the data dependence on at least one of these definitions. Thus A cannot be destination block of n if it is located before all the candidate blocks of all these instructions in D:

$$(\forall C \in \mathcal{C}(A) \exists d_i \in D : s(d_i) \in \mathcal{B}(C)) \wedge A \in \bigcap_{d_i \in D} \mathcal{B}^{\frown}(\Theta^x(d_i)) \Rightarrow A \notin \Theta^x(n) \tag{6.3.4}$$

These adapted candidate block ranges are—in combination with the relaxed assignment constraints—called the *lightweight implementation* of PR code motion since they already permit a certain amount of this transformation: They allow to schedule partial-ready copies around the candidate block ranges of dependent instructions, which is already sufficient if these ranges are small. The schedule in Fig. 6.7, for example, is already feasible in this lightweight implementation if we assume that the candidate blocks of the concurrent definitions consist only of their respective source blocks.

However, if we allow the concurrent definitions to be moved upwards via predicated code motion, then the schedule conflicts with the precedence constraints (which enforce Remark 6.3.3-(2)): For example, if $B \in \Theta^a(\texttt{Y=r1})$, then the general precedence constraint generated for $(\texttt{r1=B}, \texttt{Y=r1}) \in E_D$ and the block pair B, C is violated—it enforces $a^{\uparrow C}_{\texttt{r1=B}} = 1$, but the value of this variable is zero since a copy of `r1=B` is scheduled there (Prop. (6.3.4)). In a *precise implementation* of PR code motion, these constraints have to take Def. 6.3.2-(2) into account.

6.3.1 Precise Formulation

The realization of this is more complex; one first step is to use separated local and global precedence constraints for all dependence edges that could be ignored by PR code motion:

$$a^{\uparrow A}_n \leq a^{\uparrow A}_m \qquad \forall (m,n) \in E^{PR}_D, \forall A \in \Theta^a(m) \cap \Theta^a(n) \tag{6.3.5}$$

$$\sum_{\substack{t_n \in G(A) \\ t_n \leq t + w_{mn} - 1}} x^{At_n}_n + \sum_{\substack{t_m \in G(A) \\ t_m \geq t}} x^{At_m}_m \leq 1 \tag{6.3.6}$$

$$\forall (m,n) \in E^{PR}_D, \forall A \in \Theta^x(m) \cap \Theta^x(n), \forall t \in G_{mn}(A)$$

We have already discussed in Sec. 5.1.2 that the separated precedence constraint are equivalent to the general constraints as regards the set of integer feasible solutions, but we have also noted there that they are not as tight as the latter. The rationale behind the separation is that we have defined PR code motion at basic block granularity and not at instruction granularity: A partial-ready copy is intended to ignore dependences on instructions that are scheduled in other successor blocks (along other control flow paths), but never on instructions scheduled in the same block. We schedule around blocks, but never around individual instructions[9]. In other words, dependences are ignored globally, but not locally. Thus the local precedence constraints (6.3.6) are never violated by a partial-ready copy so that we can fully concentrate on the global variant (6.3.5), of which a far lower number is generated.

We adapt these constraints in such a way that each copy of an instruction n respects the data dependence on $d_i \in \mathsf{PR}^V(n)$ if and only if it is S_i-defined. For this purpose, we introduce additional, more differentiated a variables that characterize S_i-defined copies:

$a_n^{S_i\uparrow A} = 1 \quad \Rightarrow \quad$ An S_i-defined copy of instruction n is scheduled on each program path through $s(n)$ before A.

These new variables are added for each instruction $n \in V$ and each $S_i \in \mathsf{PR}^{\mathcal{B}}(n)$ for all blocks in $\Theta^a_{S_i}(n) := \Theta^a(n) \cap \mathcal{B}^{\prec}(S_i)$. They represent additional information about the schedule that is relevant for the precise modeling of ignorable dependences in the presence of PR code motion. In the following, we demonstrate their incorporation into the model exemplarily for an instruction n. To consider how they should be "activated", imagine that we move, starting from Ω, in a given schedule in the opposite direction of a program path $C \in \mathcal{C}(s(n))$ upwards. Along this path the conventional a_n variables are equal to one until the latest scheduled copy of n is encountered *or* until a block $S_i \in \mathsf{PR}^{\mathcal{B}}(n)$ is reached for the first time: then according to Def. 6.3.1 the next scheduled copy of n along this path will be S_i-defined; thus for the subsequent blocks along the path the variable $a_n^{S_i\uparrow A}$ should be equal to one instead of $a_n^{\uparrow A}$. This is achieved by the following constraints, of which each instance is intended to *replace* the corresponding instance of constraint (6.3.1):

$$a_n^{\uparrow S_i} \leq a_n^{S_i\uparrow A} + \sum_{t \in G(A)} x_n^{At} \qquad \forall A, S_i \in \Theta^a(n) : (A, S_i) \in E_B \wedge S_i \in \mathsf{PR}^{\mathcal{B}}(n)$$

Once equal to one, the new a_n variables are propagated in the same way as the conventional ones:

$$a_n^{S\uparrow B} \leq a_n^{S\uparrow A} + \sum_{t \in G(A)} x_n^{At} \qquad \forall A, B \in \Theta^a_S(n) : (A, B) \in E_B$$

The global precedence constraints (6.3.5) are not instantiated for edges $(m, n) \in E_D^{PR}$. Instead, the following variant is generated for the new a_n variables in order to implement Def. 6.3.2-(2), namely that an ignorable dependence on a definition $d_i \in \mathsf{PR}^V(n)$ is respected only if the copy of n is S_i-defined:

$$a_n^{S\uparrow A} \leq a_m^{\uparrow A} \qquad \begin{array}{c}\forall (m, n) \in E_D, \forall S \in \mathsf{PR}^{\mathcal{B}}(n) : S = s(m) \vee (m, n) \notin E_D^{PR}, \\ \forall A \in \Theta^a(m) \cap \Theta^a_S(n)\end{array} \tag{6.3.7}$$

[9] It would also be possible to permit a fine-grain variant of PR code motion at instruction level with multiple, differently defined copies of a use not only on a single control flow path, but even within a single basic block. We have not investigated this possibility further since it would entail a massive complexity increase (multiple sorts of x variables) that stands in no relation to the expected benefit.

In both sorts of global precedence constraints ((6.3.5) and (6.3.7)) we have to allow for the possibility that also the instruction m is subject to PR code motion. Then its global placement is possibly not only described by the $a_m^{\uparrow A}$ variables alone, but also by the new $a_m^{S_i\uparrow A}$ variables. Given an instruction n, multiple of these variables belonging to the same block A can be equal to one: For example, if $\forall i \in J : a_n^{S_i\uparrow A} = 1$ for a subset $J \subseteq \{1, \ldots, k\}$, then the copies of n scheduled on all program paths through $s(n)$ before A are S_i-defined for all $i \in J$. If such a copy is actually placed in a block A' before A, then it holds $\forall i \in J : a_n^{S_i\uparrow A'} = 0$ since Equ. (6.3.2) is also instantiated for the new a_n variables. As a result, Corollary 6.3.5 still applies if "$a_n^{\uparrow B} = 1$" is replaced by:

$$a_n^{\uparrow B} + \sum_{S \in \mathsf{PR}^{\mathcal{B}}(n) : B \in \Theta_S^a(n)} a_n^{S\uparrow B} \geq 1$$

Thus, the term $\sum_{S \in \mathsf{PR}^{\mathcal{B}}(m) : A \in \Theta_S^a(m)} a_m^{S\uparrow A}$ has to be added to the right-hand sides of Equ. (6.3.5) and (6.3.7) in order to take partial-ready copies of m into account.

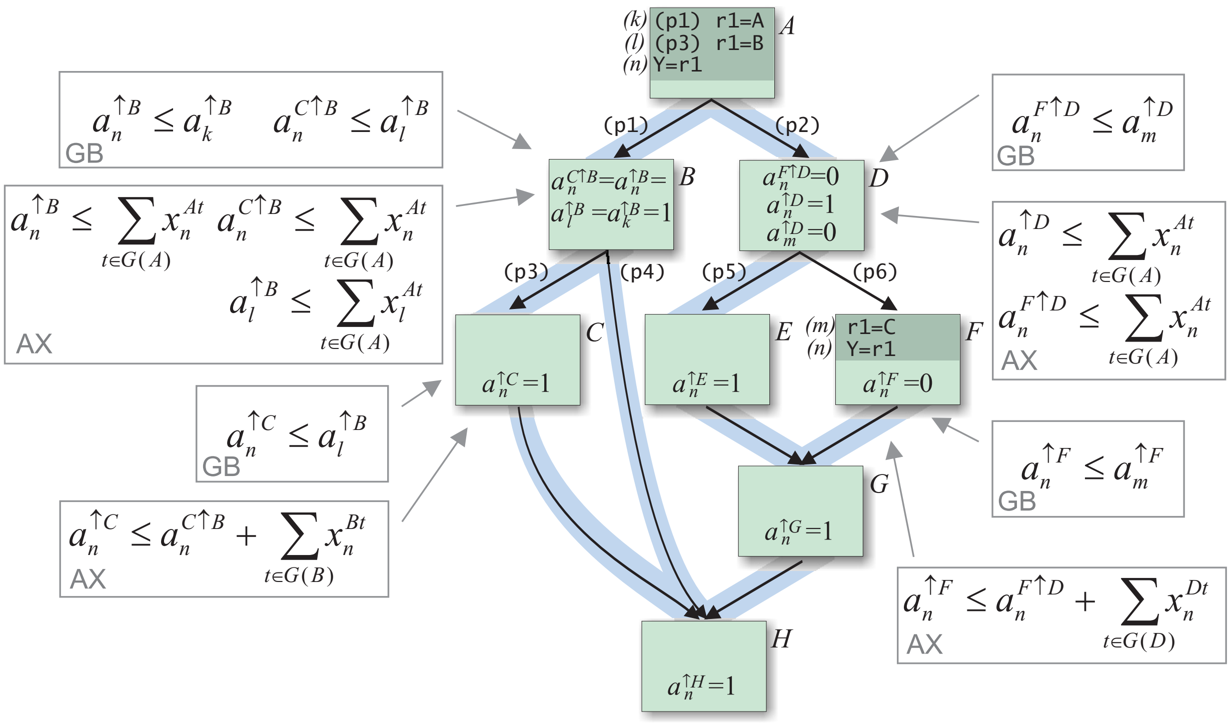

(k) (p1) r1=A
(l) (p3) r1=B
(n) Y=r1
A
(p1)
(p2)
$a_n^{C\uparrow B}=a_n^{\uparrow B}=$
$a_l^{\uparrow B}=a_k^{\uparrow B}=1$
B
$a_n^{F\uparrow D}=0$
$a_n^{\uparrow D}=1$
$a_m^{\uparrow D}=0$
D
(p3)
(p4)
(p5)
(p6)
$a_n^{\uparrow C}=1$
C
$a_n^{\uparrow E}=1$
E
(m) r1=C
(n) Y=r1
$a_n^{\uparrow F}=0$
F
$a_n^{\uparrow G}=1$
G
$a_n^{\uparrow H}=1$
H
$a_n^{\uparrow B} \leq a_k^{\uparrow B}$ $a_n^{C\uparrow B} \leq a_l^{\uparrow B}$
GB
$a_n^{\uparrow B} \leq \sum_{t\in G(A)} x_n^{At}$ $a_n^{C\uparrow B} \leq \sum_{t\in G(A)} x_n^{At}$
$a_l^{\uparrow B} \leq \sum_{t\in G(A)} x_l^{At}$
AX
$a_n^{\uparrow C} \leq a_l^{\uparrow B}$
GB
$a_n^{\uparrow C} \leq a_n^{C\uparrow B} + \sum_{t\in G(B)} x_n^{Bt}$
AX
$a_n^{F\uparrow D} \leq a_m^{\uparrow D}$
GB
$a_n^{\uparrow D} \leq \sum_{t\in G(A)} x_n^{At}$
$a_n^{F\uparrow D} \leq \sum_{t\in G(A)} x_n^{At}$
AX
$a_n^{\uparrow F} \leq a_m^{\uparrow F}$
GB
$a_n^{\uparrow F} \leq a_n^{F\uparrow D} + \sum_{t\in G(D)} x_n^{Dt}$
AX

Figure 6.9: A different application of PR code motion in the case study. The three definitions from blocks B, C, and F are abbreviated as k, l, and m, respectively.

If we assume in the case study that all definitions can be moved upwards through predicated code motion, then an application of PR code motion as in the schedule of Fig. 6.9 is possible and feasible in the developed precise formulation (but not in the lightweight version). The figure depicts the resulting variable values together with a subset of the instantiated a-x constraints (marked as AX) and global precedence constraints (marked as GB). It shows, for instance, how the C-defined, B-defined, and A-defined copy of the use in block A respects the dependences on both k and l, but ignores the dependence on m globally (since $a_n^{F\uparrow D} = 0$).

An implementation of the described modeling first identifies *candidates* for PR code motion. As mentioned, partial-ready execution occurs with speculative operands and is therefore only allowed for speculative instructions (including those added in Sec. 6.2). The set of candidates is formed of all speculative instructions with a not dominating data dependence on another instruction. If multiple of such dependences with respect to different source registers exist, then those on the closest definition is selected for PR code motion. An extension towards simultaneous PR code motion with respect to multiple source registers is straightforward, but will not be elaborated on here because the expected practical benefit is low.

When enabling PR code motion of these candidates in the model as described above, it is highly recommended to relax the a-x constraints not only for the candidates themselves, but also for depending instructions that could be speculated together with a partial-ready copy. In the above presentation, we have only concentrated on the partial-ready copies itself, but in fact the benefit from their early execution can be multiplied if together with them also data dependent instructions are scheduled earlier. In Fig. 6.7, for example, copies of an instruction that depends on `Y=r1` could be scheduled after each of the partial-ready copies in the four blocks. Such *indirectly partial-ready copies* do not directly violate a data dependence on their own, but indirectly since they are scheduled within the speculative scope of an instruction they depend on. Hence the relaxation of the a-x constraints (Def. 6.3.2-(1)) is already sufficient to make such copies feasible.

Partial-ready code motion might seem as a relatively simple variant of code motion at first sight, but the required adaptations are far-reaching and complicated, even on a semi-formal level of presentation. As indicated, they are also not without impact on the efficiency of the model. Nevertheless, it is worth the complications, as the experiments show later: there already the implemented lightweight variant proves to be highly valuable on some input programs, especially in combination with cyclic code motion, which is presented in the next section.

6.4 Cyclic Scheduling Regions and Cyclic Code Motion

Up to now, the scheduling scope is restricted to acyclic regions. This restriction is shared by many other global scheduling algorithms, however, it is hardly acceptable as it leads to a significant curtailing of the solution space: Loops are frequent, essential elements of every program with a high impact on performance. Therefore we provide in this section the necessary extensions to incorporate loops (where software pipelining is not used) and to allow code motion *into* them and *out of* them.

The movement of an instruction into a loop might sound inefficient since it is then possibly executed during each loop iteration—possibly millions of times—in relation to one single execution before. Nevertheless, this can still be beneficial if instead a nop would be executed at this place inside the loop and if it results in a decreased schedule length outside the loop. Generally, it is up to the ILP solver to decide which kinds of code motion are profitable—the model needs only to contain the options to use them.

Even if there are loops inside the scheduling region, we keep the BBG and DDG acyclic since this is a prerequisite of the so far developed ILP model: We define that backedges are *not* included in CFG and BBG—hence also *loop-carried data dependences* are not included in the DDG according to Def. 3.2.6. Loops are described separately on top of these formalisms as follows:

Definition 6.4.1 (Loop) Let an acyclic basic block graph $G_B = (\mathcal{B}, E_B, \mathcal{B}_{entry}, \mathcal{B}_{exit})$ of a scheduling region be given. A *loop* inside this region is defined by a triple $L = (\mathcal{B}_L, E_L^{\Lsh}, H_L)$ and comprises all the blocks in the subset $\mathcal{B}_L \subseteq \mathcal{B}$, which is termed the *loop body*. It must satisfy the following two conditions:

- The *loop header* $H_L \in \mathcal{B}_L$ is the unique entry point of the loop[10]; it dominates all blocks $\mathcal{B}_L$ in G_B.
- For each block in $\mathcal{B}_L$, there is a path in the graph $(\mathcal{B}_L, E_B \cup E_L^{\Lsh})$ leading from this block—along one of the *backedges* in $E_L^{\Lsh} \subseteq \mathcal{B}_L \times \{H_L\}$—back to H_L.

The *loop exit edges* are given by $E_L^{\downarrow} := E_B \cap \mathcal{B}_L \times (\mathcal{B} \setminus \mathcal{B}_L)$. The block sets of two different loops are either disjoint or one is included in the other. In the latter case, the contained loop is either *nested* within the other (if both have different headers) or termed a *subloop* of the other (if both have the same header). Loops that do not contain nested loops are called *inner loops* or *innermost loops*. □

The concept of subloops arises from loops with multiple backedges: then we can associate to each backedge a loop of its own, i.e., with only this backedge. We assume in the following that given loops are *maximal* with respect to $\mathcal{B}_L$ so that they are no subloops of other loops. Fig. 6.10 gives an example of an inner loop where the blocks B and D form a subloop.

Backedges do not alter the dominance or postdominance relationships, hence these can be computed either on $(\mathcal{B}, E_B)$ or on $(\mathcal{B}, E_B \cup E_L^{\Lsh})$. When determining reaching definitions and concurrent definitions, however, it is important to allow for the possibility that a definition reaches a use via a backedge: In Fig. 6.10, for example, the definitions `op`$_2$ `r1=..` and `op`$_3$ `r1=..` *are* concurrent—even without the load in G—since the `op`$_3$ reaches the use `op`$_4$ `r2=r1`, too, via a backedge. Thus, they must be considered as non-speculative instructions (executing `op`$_3$ `r1=..` in B speculatively, for instance, would apparently change the program semantics).

The presence of loops can also lead to the phenomenon that a definition is concurrent *with itself*, or more precisely, that different executions of it in different loop iterations are concurrent. This can be the case for a definition n

- if its source block $s(n)$ inside a loop does not postdominate the loop header and

[10]Loops with this property are termed *natural loops* [Muc97]. They lead to a *reducible* control flow graph where we can identify backward edges by the property that they lead from a block to another one that dominates the former. Loops with multiple entry block are rare [Muc97], hence we do not address here how to deal with them.

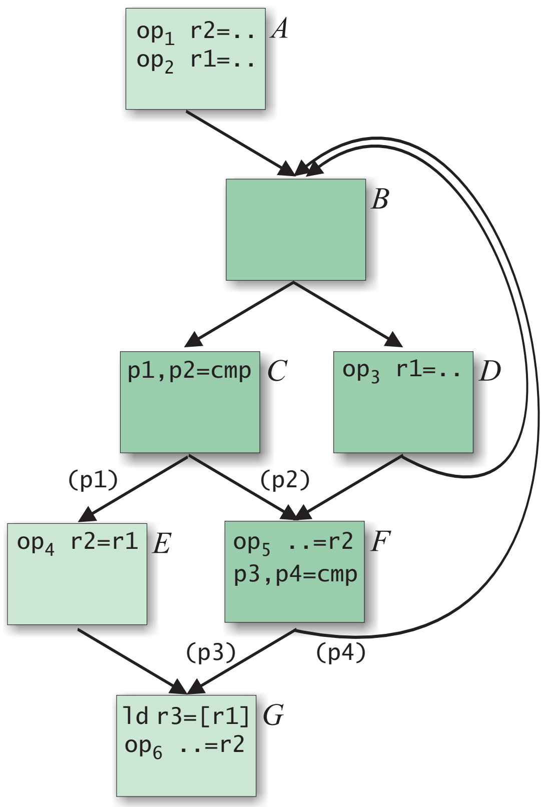


Figure 6.10: Case study of loops.
The loop consists of the blocks B, C, D and F, has header B, backedges (D, B) and (F, B) and loop exit edges (C, E) and (F, G).

- if it reaches a use with source block outside the loop, or inside the loop but not dominated by $s(n)$.

For example, the $\mathtt{op}_3$ `r1=..` in Fig. 6.10 would have to be considered as a concurrent definition (and thus non-speculative) even without the other definition in block A.

There are further, minor complications due to cyclic regions in combination with the previously presented extensions: Predicate register initializations due to predicated downward code motion within a loop must be inserted in a dominating block *inside* the loop body (like the loop header). In Sec. 6.1.2, we predefine that BBG backedges are *never* fall-through edges to avoid cycles of such edges in the schedule that could not be realized by the basic block order in memory.

Apart from these adaptations, the main consequences of loops inside the scheduling region are restrictions with respect to code motion. The general rule that non-speculative instructions must not be moved into blocks that are not postdominated or dominated by its source block applies also in the presence of loops. But in addition to this rule, there are further exclusion criteria that are discussed in the following two subsections.

6.4.1 Code Motion into Loops

In Fig. 6.10, the mentioned general rule does not exclude the possibility to move the load `ld r3=[r1]` upwards out of G into the loop since all blocks in the loop body are postdominated by G. There it is executed only if it would also be executed in G since the control will eventually exit the loop and flow through this block. But this view is too shortsighted: The load could be executed many times prior to this exit with undefined operands (written by $\texttt{op}_3$ `r1=..`)—in short, under conditions that must be regarded as speculative.

This leads to a more differentiated conception of control speculation in the presence of loops: each instruction that is moved *upwards* into a loop is scheduled speculatively *with respect to the loop exit* since it is useful only in iterations where this exit is taken. This is then problematic if the instruction is non-speculative and if one of its operands is written inside the loop—then an execution with different operands than in the original program could occur. Hence upward code motion into loops is prohibited under this condition for non-speculative instructions like normal loads, stores, speculation checks, and compares.

A speculative version of the load in Fig. 6.10, however, could in principle be scheduled into the blocks C and D inside the loop. But the movement of control speculative loads into loops should be dealt with care, especially if its address operand is written there: then executions in different loop iterations with different addresses could pollute the TLBs and the cache. Thus we forbid the movement into such loops. Otherwise, if the address operand is loop invariant, then the broad rule from the end of Sec. 6.2.1 still applies that prevents the movement of speculative loads into relatively much more frequently executed destination blocks. This excludes moving them into loops with a relatively high trip count.

Furthermore, definitions must not be moved into a loop that contains another definition that is concurrent with it. For instance, in Fig. 6.10 the instruction $\texttt{op}_2$ `r1=..` must not be moved into the loop (as it is concurrent with $\texttt{op}_3$ `r1=..`). Concurrent definitions must also not be moved upwards into a loop that contains a use of the value they write—an example of this is the concurrent definition $\texttt{op}_4$ `r2=r1` (due to the use $\texttt{op}_5$ `..=r2`). However, they *may* be moved *downwards* into a loop that contains such a use[11]—this applies to the statement $\texttt{op}_1$ `r2=..` in Fig. 6.10. In addition to these restrictions, instructions that are not multiply executable (as characterized in Sec. 6.3) can generally not be moved into loops.

Apparently, all described limitations do not only apply to code motion into a loop, but also to code motion into a *subloop* within the same loop. The same argumentation that restricts the movement into loops holds for subloops, too. However, there is one condition that renders all the listed exclusion criteria void: if the instruction is guarded inside the (sub-)loop by a predicate that is true only on (sub-)loop exit (or another predicate that is a logical implication of it). Thus, when the candidate block ranges of instructions are extended via predication as described in Sec. 6.1, loops impose no limits (but the dependences on the controlling compares that are introduced do).

[11]This is possible since the movement occurs across the loop header that dominates all blocks in the loop—but even then only if none of the other exclusion criteria apply.

The prerequisites for code motion *across* loops are more restrictive. We can—similarly to wavefront scheduling in Sec. 3.3.1—regard a loop as a single instruction that summarizes all data flow information related to the loop. Then another instruction can only be moved across the loop if it is not data dependent on this imaginary instruction. This is the case if none of its destination operands is read or written inside the loop. Additionally, none of its source operands may be written there.

6.4.2 Cyclic Code Motion out of Loops

Code motion out of loops if trivial if the code is loop invariant [SS02]—for such instructions, the loop boundaries impose no limits. For instructions like the "`op`$_1$ `rX=rU`" in Fig. 6.11 whose input operand is written inside the loop body, however, passing the loop boundary is more difficult: Such an instruction can be hoisted upwards out of the loop if it is not only moved into predecessors of the loop header, but also *along each backedge* to the bottom blocks of the loop and their predecessors (illustrated in Fig. 6.11 (b)). We call this kind of code motion *cyclic* or *circular* [Jai91].

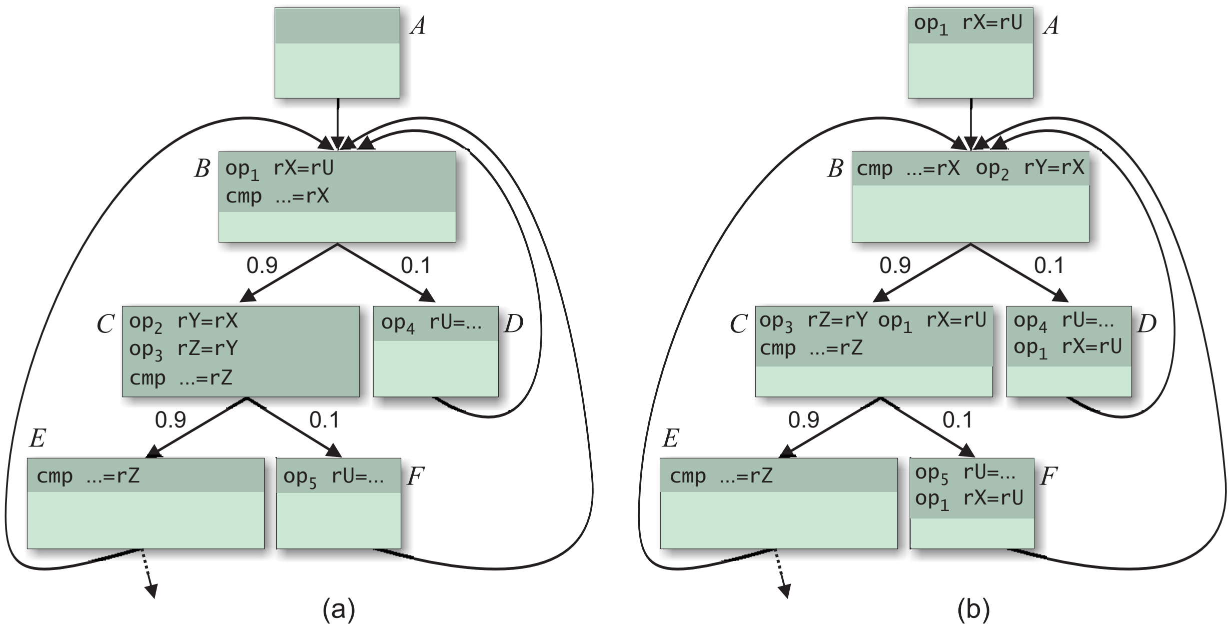


Figure 6.11: Case study of cyclic code motion. In (a), each instruction is scheduled in its source block. (b) demonstrates cyclic code motion of "`op`$_1$ `rX=rU`".

Copies that are moved in such a way in the opposite direction of the backedges are effectively scheduled in the previous loop iteration. This means that they are (possibly) executed speculatively since they are only useful if the control takes one of the backedges, but not if it exits the loop via one of the loop exit edges. Thus we consider cyclic code motion generally as speculative. It can be profitable if the cyclically moved code is overlapped inside the loop body with instructions from the previous iteration, reducing the critical path length here. In Fig. 6.11 (b), it reduces the length of the hot subpath B-C-E inside the loop from 6 to 4 cycles (of which one cycle is saved by moving "$\texttt{op}_1$ rX=rU" upwards to block B).

Fig. 6.12 (a) shows that we can go even further and move also "$\texttt{op}_2$ rY=rX" to the previous loop iteration, reducing the length to 3 cycles. As with the former instruction, four compensation copies have to be scheduled, one in the predecessor of the loop header and three in different blocks in the loop body. It is noteworthy about these copies that the cyclic copy of "$\texttt{op}_1$ rX=rU" in C is partial-ready since it ignores the dependence on the "$\texttt{op}_5$ rU=..." along the subpath B-C-F.

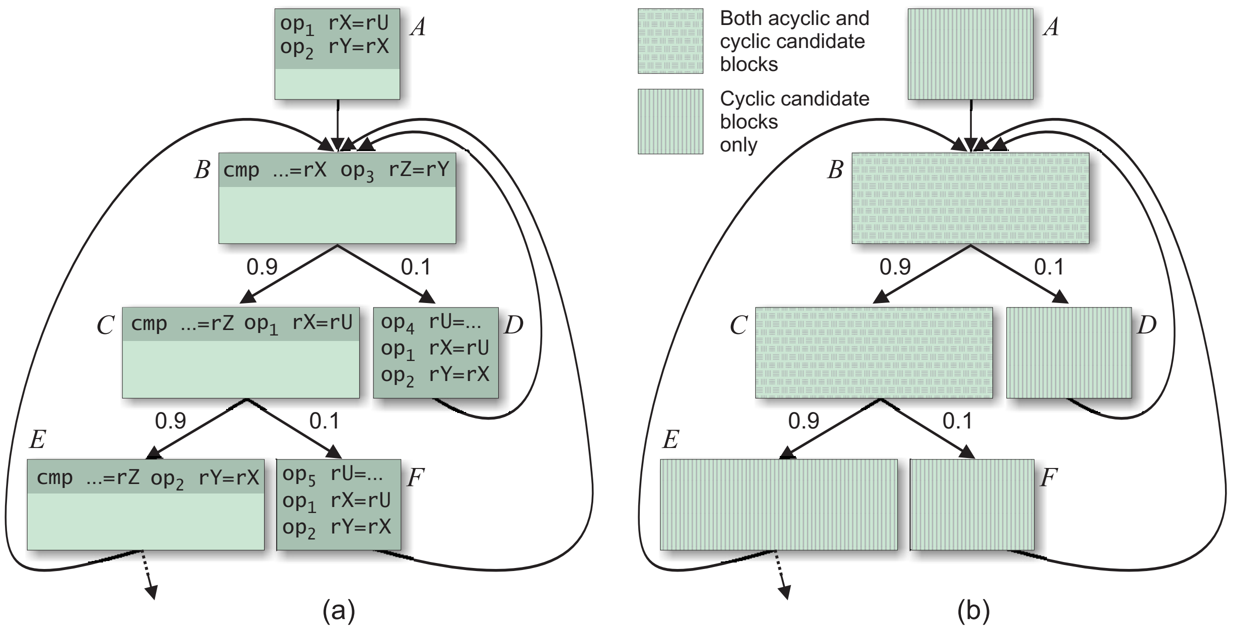


Figure 6.12: Moving also "$\texttt{op}_2$ rY=rX " cyclically (a). (b) shows the candidate blocks of this instruction.

Cyclic code motion can be regarded as a simple variant of software pipelining [SS02]. In general, the latter scheduling technique is the first choice for loops, also because it is well supported by the architecture [Int02a]. However, in practice there are still many loops where the use of software pipelining is difficult or even disadvantageous [HP03]:

1. If they contain *other loops*. Although recent work shows that software pipelining can be performed at an arbitrary level of nesting [RDGG04], implementations like Intel's Itanium compiler typically consider only innermost loops for software pipelining [Int03].

2. If they contain *procedure calls*. Ways to incorporate these branches into software-pipelined loops are thinkable, but even advanced compilers do not implement them [Int03]. At least small procedures can be inlined into the loop.

3. If they contain *complex control flow*. This limitation is more fundamental than the previous two. Loops with control flow can be converted to predicated straight-line code via if-conversion, however, the resulting initiation interval is the worst case over all control flow paths through the loop body. To avoid this, there exist complicated, code-expanding techniques that generate multiple kernels with variable initiation intervals for different paths [SL96, Lav97]. Intel's Itanium compiler removes control flow from the loop body prior to software pipelining using a combination of if-conversion and tail duplication [MCWL01].

4. If they have *low trip counts*. Software pipelining increases the throughput of loop iterations by exploiting parallelism between them, but the time needed to complete a single loop iteration may increase [Int02a]. If the trip count is low, the increased throughput may not amortize the increased latency.

When software pipelining does not succeed, cyclic code motion can help alleviate the inefficiencies due to the static scheduling of the loop; however, it should be used with care since it comes at the price of code expansion. In contrast to software pipelining, it allows to overlap instructions from the first loop iteration with code before the loop. This can be regarded as an *instruction-wise* application of *loop peeling*, a transformation that unrolls the first loop iteration and schedules it before the loop [Muc97].

We have integrated upward[12] cyclic code motion into the ILP for speculative instructions[13] (including those added in Sec. 2.1.5) out of the innermost loop that contains the instruction's source block. Let $V^L_{\circlearrowleft} \subseteq V$ denote those instructions that satisfy these conditions with respect to a loop L and that have been selected as candidates for cyclic code motion (more on the selection at the end of the section). In the following, we demonstrate the necessary changes exemplarily considering an instruction $n \in V^L_{\circlearrowleft}$.

At first, we define the destination block candidates of cyclic code motion. The previous candidate blocks are termed from now on *acyclic* candidate blocks to distinguish them from these new *cyclic* candidate blocks. The *potential cyclic candidate blocks* $\Theta_{\circlearrowleft}(n)$ are composed of the predecessors of the loop header, $\Theta_{\circlearrowleft U}(n) := \mathcal{B}^{\prec}(H_L)$, and the loop body $\Theta_{\circlearrowleft B}(n) := \mathcal{B}_L$. The previous potential candidate blocks (5.3.1) become the new potential *acyclic* candidate blocks, $\Theta_{\updownarrow}(n)$. The new set $\Theta(n)$ is defined as the union of $\Theta_{\circlearrowleft}(n)$ and $\Theta_{\updownarrow}(n)$:

$$\Theta_{\circlearrowleft}(n) := \Theta_{\circlearrowleft U}(n) \cup \Theta_{\circlearrowleft B}(n) \qquad \Theta(n) := \Theta_{\circlearrowleft}(n) \cup \Theta_{\updownarrow}(n) \tag{6.4.1}$$

According to Def. 5.3.1, we define also the (actual) *cyclic candidate blocks* $\Theta^x_{\circlearrowleft U}(n) \subseteq \Theta_{\circlearrowleft U}(n)$ and $\Theta^x_{\circlearrowleft B}(n) \subseteq \Theta_{\circlearrowleft B}(n)$ for which x_n variables are instantiated. The relationships between the sets

- $\Theta^x(n)$, $\Theta^x_{\updownarrow}(n)$, $\Theta^x_{\circlearrowleft}(n)$, $\Theta^x_{\circlearrowleft U}(n)$, and $\Theta^x_{\circlearrowleft B}(n)$, as well as the sets
- $\Theta^a(n)$, $\Theta^a_{\updownarrow}(n)$, $\Theta^a_{\circlearrowleft}(n)$, $\Theta^a_{\circlearrowleft U}(n)$, and $\Theta^a_{\circlearrowleft B}(n)$

[12]The downward variant in the forward direction of the backedges would be possible, too.

[13]*Predicated* cyclic code motion for non-speculative instructions is also thinkable. However, we have not followed this up since the impact of predicated code motion in general is low (see Sec. 7.3.2).

are analogous to above (6.4.1). The three properties of Def. 5.3.1 are enforced for the sets $\Theta_{\circlearrowleft U}(n)$, $\Theta^x_{\circlearrowleft U}(n)$, $\Theta^a_{\circlearrowleft U}(n)$, and $\Theta_{\circlearrowleft B}(n)$, $\Theta^x_{\circlearrowleft B}(n)$, $\Theta^a_{\circlearrowleft B}(n)$, respectively, but not for the combined sets $\Theta(n)$, $\Theta^x(n)$, and $\Theta^a(n)$, which merge both acyclic and cyclic candidate blocks.

Fig. 6.12 (b) depicts the candidate blocks of instruction $\mathtt{op}_2$ in the case study. These are $\Theta^x_{\updownarrow}(\mathtt{op}_2) = \{B, C\}$, $\Theta^x_{\circlearrowleft U}(\mathtt{op}_2) = \{A\}$, and $\Theta^x_{\circlearrowleft B}(\mathtt{op}_2) = \{B, C, D, E, F\}$. This shows that the sets $\Theta^x_{\updownarrow}(n)$ and $\Theta^x_{\circlearrowleft B}(n)$ can *overlap* so that there exist candidate blocks in which the same instruction can be placed either acyclically or cyclically. This raises the fundamental question whether the same or separate x variables should be used to model both (mutually exclusive) placements. On the one hand, using separate variables—like $x^{At}_{\updownarrow n}$ and $x^{At}_{\circlearrowleft n}$—leads to a more straightforward, simpler model. This is because acyclic and cyclic copies of the same instruction differ in many ways, also with regard to data dependences (as discussed below). On the other hand, the duplication can effectively double the number of these variables, which has inevitably a negative impact on the solution times. Therefore we have decided to use the *same x and a variables* to model both acyclic and cyclic scheduling.

As a decision variable for cyclic code motion, the a_n variable of the loop header, $a_n^{\uparrow H_L}$, is a suggestive choice. It is defined that if this variable is equal to one, then all copies of n are scheduled cyclically (in blocks of $\Theta^x_{\circlearrowleft}(n)$) and else acyclically (in blocks of $\Theta^x_{\updownarrow}(n)$). These two alternatives are also referred to as the two different *states* (w.r.t. cyclic code motion) of the instruction. If $a_n^{\uparrow H_L}$ and $1 - a_n^{\uparrow H_L}$ are zero, then all the x_n/a_n variables belonging to candidate blocks that are *only* cyclic and acyclic, respectively, must be zero, too. This is explicitly enforced by the following constraints:

$$\begin{array}{rcl} a_n^{\uparrow A} & \leq & a_n^{\uparrow H_L} \\ \sum\limits_{t \in G(B)} x_n^{Bt} & \leq & a_n^{\uparrow H_L} \\ \multicolumn{3}{l}{\forall A \in \Theta^a_{\circlearrowleft}(n) \setminus \Theta^a_{\updownarrow}(n),} \\ \multicolumn{3}{l}{\forall B \in \Theta^x_{\circlearrowleft}(n) \setminus \Theta^x_{\updownarrow}(n)} \end{array} \qquad \begin{array}{rcl} a_n^{\uparrow A} & \leq & 1 - a_n^{\uparrow H_L} \\ \sum\limits_{t \in G(B)} x_n^{Bt} & \leq & 1 - a_n^{\uparrow H_L} \\ \multicolumn{3}{l}{\forall A \in \Theta^a_{\updownarrow}(n) \setminus \Theta^a_{\circlearrowleft}(n),} \\ \multicolumn{3}{l}{\forall B \in \Theta^x_{\updownarrow}(n) \setminus \Theta^x_{\circlearrowleft}(n)} \end{array}$$

The assignment constraints (5.3.10) of n are adapted to cyclic code motion as follows:

$$a_n^{\uparrow A} + \sum_{t \in G(A)} x_n^{At} + a_n^{\uparrow H_L} \geq 1 \qquad \forall A \in \widetilde{\Theta^a_{\updownarrow}}(n)$$

This inequality ensures that *either* cyclic code motion is switched on via $a_n^{\uparrow H_L} = 1$, *or* that otherwise n is scheduled acyclically like a normal instruction: if $a_n^{\uparrow H_L} = 0$, then the constraint acts like previously Equ. (5.3.10) (note the use of $\widetilde{\Theta^a_{\updownarrow}}(n)$ instead of $\widetilde{\Theta^a}(n)$). The "$\geq$" in place of "$=$" is only necessary if $A \in \Theta^a_{\circlearrowleft B}(n) \cap \Theta^a_{\updownarrow}(n)$—then due to the double use of the x variables the left-hand side can grow larger than one if $a_n^{\uparrow H_L} = 1$ (e.g., consider in the case study the instance created for $C \in \widetilde{\Theta^a_{\updownarrow}}(\mathtt{op}_2)$).

It remains to model the effect of $a_n^{\uparrow H_L} = 1$, namely the actual cyclic scheduling of n. This is implemented by instantiating the a-x constraints also for all BBG edges of the form (A, H_L), including backedges:

$$a_n^{\uparrow B} = a_n^{\uparrow A} + \sum_{t \in G(A)} x_n^{At} \qquad \forall B, A \in \Theta^a(n) : (A, B) \in E_B \setminus E_n^{\times} \cup E_L^{\Lsh} \tag{6.4.2}$$

In doing so, we model that $a_n^{\uparrow H_L} = 1$ implies that a copy of n is scheduled on each program path through $s(n)$ before H_L (in a block from $\Theta^x_{\circlearrowleft U}(n)$) *and* additionally on each cyclic subpath that leads from H_L—along one of the backedges—back to H_L (in blocks from $\Theta^x_{\circlearrowleft B}(n)$). In the case study, four of these equations are instantiated for $n = \texttt{op}_1$ and $B = H_L$:

$$\begin{aligned}
a_{\texttt{op}_1}^{\uparrow H_L} &= a_{\texttt{op}_1}^{\uparrow A} + \sum_{t \in G(A)} x_{\texttt{op}_1}^{At} \\
a_{\texttt{op}_1}^{\uparrow H_L} &= a_{\texttt{op}_1}^{\uparrow E} + \sum_{t \in G(E)} x_{\texttt{op}_1}^{Et} \\
a_{\texttt{op}_1}^{\uparrow H_L} &= a_{\texttt{op}_1}^{\uparrow F} + \sum_{t \in G(F)} x_{\texttt{op}_1}^{Ft} \\
a_{\texttt{op}_1}^{\uparrow H_L} &= a_{\texttt{op}_1}^{\uparrow D} + \sum_{t \in G(F)} x_{\texttt{op}_1}^{Dt}
\end{aligned}$$

When generating these equations (6.4.2), it is important that the special variable $a_n^{\uparrow H_L}$ never appears on their right-hand side (i.e., that it is omitted if $A = H_L$)—otherwise they could form a cycle. Moreover, the possible double use of the x and a variables in some blocks in $\Theta^a(n)$ implies that instances of Equ. (6.4.2) hold in the acyclic as well as in the cyclic state—but this is not necessarily true: If $B \in \Theta^a_{\circlearrowleft}(n) \setminus \Theta^a_{\updownarrow}(n)$ and $A \in \Theta^a_{\updownarrow}(n) \cap \Theta^a_{\circlearrowleft}(n)$, for example, then the inequality should have no effect in the acyclic state because B is then no candidate block. Since we have not differentiated between the two states with respect to the variables, we must now do so with respect to constraints.

For this purpose, we check for each instance of (6.4.2) whether it *would have been instantiated* within the candidate block sets $\Theta^a_{\circlearrowleft}(n)$[14] and $\Theta^a_{\updownarrow}(n)$ (instead of $\Theta^a(n)$), too. If this would have occurred in neither case, then it is omitted. If it would be instantiated only within *one* of the sets $\Theta^a_{\circlearrowleft}(n)$ and $\Theta^a_{\updownarrow}(n)$, then a version must be generated that is *cyclic only* and *acyclic only*, respectively, which means that it may only be effective in these states. For the latter case, such a version consists of the following two inequalities in place of (6.4.2):

$$\begin{aligned}
a_n^{\uparrow B} &\leq a_n^{\uparrow A} + \sum_{t \in G(A)} x_n^{At} + a_n^{\uparrow H_L} \\
a_n^{\uparrow H_L} + a_n^{\uparrow B} &\geq a_n^{\uparrow A} + \sum_{t \in G(A)} x_n^{At}
\end{aligned}$$

These two inequalities are equivalent to (6.4.2) if $a_n^{\uparrow H_L} = 0$ and void (always satisfied) otherwise. For cyclic only instances, $a_n^{\uparrow H_L}$ is replaced by $1 - a_n^{\uparrow H_L}$. An example of a cyclic only instance in Fig. 6.12 is the a-x constraint created for $\texttt{op}_2$ and the BBG edge (B, D).

[14]In this case, $E_n^{\times}$ is omitted in Equ. (6.4.2) since the source block of n in the cyclic state is effectively H_L.

The last remaining constraints that require modifications due to cyclic code motion are the precedence constraints (5.3.12). The cyclic state of an instruction can introduce new data dependences and nullify others: Consider a dependence $(m, n) \in E_D$ between two instructions that can both be cyclically moved (with respect to the same loop). If *only* m is scheduled in this way, then the dependence should be ignored: then m is scheduled in the previous loop iteration and therefore executed before n anyway. In other words, then the dependence does not impose an *order* on the scheduled copies of m and n within the loop body since they are semantically in different loop iterations (consider, for instance, the copies of $\texttt{op}_1$ and $\texttt{op}_2$ in Fig. 6.11 (b), whose order does not reflect the true dependence from the former instruction to the latter). If also n is in the cyclic state so that *both* instructions are scheduled in the previous loop iteration, however, then the order must allow for the dependence again (as in Fig. 6.12 (a)). The lifting of the precedence constraints under the described condition is implemented by adding the relaxer $a_m^{\uparrow H_L} - a_n^{\uparrow H_L}$ to the right-hand side of Equ. (5.3.12). It nullifies these constraints if m is in the cyclic state and n not.

An instruction in the cyclic state must also allow for loop-carried data dependences. Given a loop L, these dependences are collected in the set $E^L_{\circlearrowleft D} \subseteq V \times V$ (as mentioned, they are not included in the DDG); they are identified like normal data dependences according to Def. 3.2.6 except that the control flow paths in the definition are required to traverse exactly one backedge of the loop. Thus an edge $(m, n) \in E^L_{\circlearrowleft D}$—termed a ***DDG backedge***—describes a dependence of n executed in loop iteration i on m executed in loop iteration $i - 1$. It must *only then* take effect in the schedule—i.e., determine the order of m and n—if n is moved cyclically to the previous loop iteration and m not.

This is implemented by adding precedence constraints for all DDG backedges and providing them with the relaxer $1 + a_m^{\uparrow H_L} - a_n^{\uparrow H_L}$. Examples of DDG backedges in the case study are $(\texttt{op}_4, \texttt{op}_1)$, $(\texttt{op}_5, \texttt{op}_1)$ (true), and $(\texttt{op}_2, \texttt{op}_1)$ (anti). WAR backedges (as the latter) typically emerge as the counterparts of normal RAW dependences (here $(\texttt{op}_1, \texttt{op}_2) \in E_D^{RAW}$). In this case, the WAR backedge $(\texttt{op}_2, \texttt{op}_1)$ forbids in Fig. 6.11 (b) to move the copy of $\texttt{op}_1$ in C upwards before the copy of $\texttt{op}_2$ in block B (which would in fact alter the semantics).

In both relaxers, $a_m^{\uparrow H_L}$ and $a_n^{\uparrow H_L}$ are replaced by zero if $m \notin V^L_{\circlearrowleft}$ and $n \notin V^L_{\circlearrowleft}$, respectively. In addition, they can be simplified in those instances of Equ. (5.3.12) where $A \in \Theta^a_{\circlearrowleft}(n) \setminus \Theta^a_{\updownarrow}(n)$: In any solution that violates such an instance, the sum of the first two terms on its left-hand side must be equal to one. This implies that $a_n^{\uparrow H_L}$ is equal to one (see page 171) so that it can be directly replaced by this constant. In an analogous way, it can be substituted by zero if $A \in \Theta^a_{\updownarrow}(n) \setminus \Theta^a_{\circlearrowleft}(n)$ and $a_m^{\uparrow H_L}$ by the constants one and zero if $A, B \in \Theta^a_{\circlearrowleft}(m) \setminus \Theta^a_{\updownarrow}(m)$ and $A, B \in \Theta^a_{\updownarrow}(m) \setminus \Theta^a_{\circlearrowleft}(m)$, respectively.

A further complexity reduction is possible by minimizing DDG backedges analogously to Def. 3.2.7 and by taking them into account when reducing the candidate block ranges. As mentioned earlier, Equ. (5.3.2) is also applied to the set $\Theta^x_{\circlearrowleft B}(n)$ in order to remove candidate blocks that are impossible due to data dependences. This process can be extended towards loop-carried dependences to limit upward code motion of cyclic copies inside the loop body. The following variant of (5.3.2) excludes impossible candidate blocks due to a DDG backedge $(m, n) \in E^L_{\circlearrowleft D}$

(applicable only if $m \notin V^L_{\circlearrowleft}$):

$$A \in \Theta(m) \cap \mathcal{B}^{\frown}(\Theta^x(m)) \Rightarrow A \notin \Theta^x_{\circlearrowleft B}(n) \tag{6.4.3}$$

As explained in Sec. 6.3, the application of (5.3.2) (and its above variant (6.4.3)) is too restrictive and thus not permissible in combination with PR code motion. Then versions of Equ. (6.3.3) and (6.3.4) must be used instead of (5.3.2) and (6.4.3). For the sake of simplicity, we allow PR code motion for instructions from $V^L_{\circlearrowleft}$ only in the cyclic state[15] and only with respect to dependences that are loop-carried (which is no restriction since all other dependences are dominating in the cyclic state and thus cannot be ignored).

The candidate set $V^L_{\circlearrowleft}$ is made up of all speculative instructions that are loop variant and not data dependent on another instruction in the same loop that is not eligible for cyclic code motion. For efficiency reasons, this set should be as small as possible. Future implementations could exclude candidates for which it can be shown that moving them cyclically is counterproductive since it would yield (due to DDG backedges) a longer schedule than achievable otherwise. For this purpose, it is possible to schedule the loop body heuristically in order to obtain an upper bound on its minimal global schedule length. Then cyclic code motion of an instruction can be excluded from the search space if a lower bound (e.g., critical path length) shows that any schedule of the loop body resulting from this would exceed the upper bound in length.

6.5 Subsequent Optimization Phases

A consequence of the used objective function is that the ILP solver has a blind spot for everything that is not related to the schedule length. This can result in schedules that are suboptimal with respect to other aspects like register usage. To compensate for this we can perform subsequent optimizations *while preserving the minimal schedule length.* This can be done by solving a second ILP subsequently that is the same as the first one, except that it has a different objective function and that the length of each block is fixed to its solution value of the first phase (this is achieved by adding respective equations).

In doing so, we optimize for lower priority goals within the space of optimal-length schedules[16]. We briefly sketch some possible objectives for the second phase; only the first one has been implemented and is currently used in the experiments.

- **Minimization of the instruction count**: Nothing detains the ILP solver from using more speculation and more compensation copies than necessary, as long as the resulting schedule has minimal length. To deal with this indifference, we use a new objective function during the second phase that minimizes the number of scheduled instructions (i.e., the sum of all

[15]This continues to ensure that acyclic and cyclic copies of the same instruction never coexist in a schedule.

[16]Strictly speaking, this claim is not correct since we fix not the global schedule length (Equ. (5.3.16)), but the individual block lengths (the variables B^A_t). This is more restrictive as there could be different block length combinations with the same global schedule length. This limitation is accepted because it is minor and leads to very fast solution times in the second phase. It is in later phases also possible to provide the known solution from the first phase as a start solution to the ILP solver.

x_n^{At} variables, including those for branch instructions). We also add all the variables S_n^D and S_n^{DA} to this objective function to remove unnecessary usages of data speculation and to avoid the more expensive advanced load check. In the experiments (see Sec. 7.2), this takes not more than one second for all input programs and reduces the instruction count by about 5% on average.

- **Reducing register pressure**: Long-range code motion leads to long live ranges and increases thereby the register pressure. A subsequent phase could alleviate this by minimizing the distances between definitions and their last uses.

- **Stall minimization** aims to minimize the stalls caused by cache misses. In the first phase, we assume that every load is an L1 cache hit which delivers data after one cycle. If a load misses the L1 cache, however, then the processor stalls at the first use of the loaded value (if it is not yet available). The resulting stall duration is inversely proportional to the load-use distance in the schedule. Hence the second phase could reduce potential stall cycles by expanding these distances. This involves a reordering of instructions, utilizing slack in the schedule (cf. *balanced scheduling* [KE93]).

The latter two objectives are conflicting in the sense that the first one aims to shrink live ranges while the second one aims to expand (some of) them. Because of the large impact of stalls on performance, the second one should be given a higher priority.

Measuring the distance between two instructions in the schedule, as needed for stall minimization, is a challenging problem to model: This distance depends on which control flow subpath is taken between the blocks where the two instructions are scheduled. To take stalls precisely into account, distances should be weighted by the execution frequencies of the respective subpaths. A possible approach to this will be outlined in Sec. 6.7.

6.6 Bundling

The schedule delivered by the ILP solver consists of a series of instruction groups for each basic block. These groups still have to be integrated into a bundle structure as described in Sec. 2.1.1.2. This involves the selection of a template (2.1.1) for each bundle, which assigns one of the execution unit types M, I, F, and B to each of its three slots (numbered 0,1, and 2). The instructions then must be placed into slots of suitable types while preserving possible intra-group data dependences. These dependences give the placement the characteristics of a scheduling problem.

The primary goal during bundling is to avoid any occurances of *split issue*. This term denotes the condition when an instruction group is split up dynamically into more than one issue group, resulting in extra cycles (see Sec. 2.2.3.2). Such a departure from the computed schedule should be completely avoided—each instruction group should be bundled in such a way that it can be issued to the processor's execution units in one cycle, as intended. This requires, inter alia, that it does not extend over more than two bundles.

A secondary, but also important goal is to minimize the number of nops. One of the main sources of unused slots are the stops, which must be inserted to delimit instruction groups. They can only be placed between bundles and—for two templates—also within a bundle. Since the instruction group sizes are naturally often not multiples of three, intra-bundle stops play a pivotal role in minimizing the number of nops. However, they also lead to interdependences between successive instruction groups during bundling: If a group ends inside a bundle, then the next instruction group automatically starts at the next slot within the same bundle (of the same template). In this way, intra-bundle stops can influence bundling decisions of adjacent instruction groups, an effect that can even propagate over the whole bundle sequence. As intra-bundle stops can only be used together with the templates MMI and MII (namely after slots 0 and 1 there, respectively), they also interact with the template selection.

Due to these interdependences it is not advisable to tackle bundling on a per-group basis. Our approach, which is based on *precomputed results* and *dynamic programming* [KW01, Win01], bundles basic blocks as a whole. It distinguishes between two closely coupled subproblems: *micro-scheduling*, which refers to the problem to find a *partial* bundle sequence for a single instruction group, and *sequencing*, the phase that combines the partial sequences to form a sequence for the whole basic block. Under the assumption that the given instruction groups are fixed (i.e., no instructions can be interchanged between them), it computes bundle sequences of minimal size or, in other words, with a minimal number of nops.

Let for a basic block the instruction groups of the respective cycles be given by $G_1, \ldots, G_T \subseteq V$. The set of the last cycle, G_T, includes all outgoing branches due to Sec. 6.1.2. In order to obtain a minimal-length bundle sequence for the whole block, we consider *concatenations* of *partial bundle sequences* that span only a certain range of these instruction groups. Such partial sequences may also start and end within a bundle: We say that it *starts at slot* 0,1 or 2 if its first slot has this number (in the last two cases, its first bundle must be MMI and MII, respectively, and is only partially used). It is said to *end at slot* 0,1 or 2 if it ends just *before* a slot with this number in a bundle. In the last two cases, its last bundle is MMI and MII, respectively, and is only partially used. Fig. 6.13 depicts for an instruction group given in the box partial sequences with all possible start/end slot combinations.

Evidently, two partial sequences s and t can be concatenated only if the end slot number of s and the start slot number of t are equal. The concatenation is denoted by $s \oplus t$. We use the following sets to classify partial sequences:

$S_{xy}(G_i, G_j)$	$=$	The set of all possible (partial) bundle sequences comprising instruction groups $G_i, \ldots, G_j$, starting at slot x and ending at slot y.

We say that $j - i + 1$ is the *size* of the partial bundle sequence (the number of instruction groups it comprises) and use the attributes “smaller” and “larger” in this context. In contrast, its *length* (“shorter”, “longer”) is defined as the number of bundles it consists of (including partially used bundles). Let $\|s\|$ denote this length of a sequence s. Then it holds for the length of the concatenation of two partial sequences $s_{xu}(G_i, G_k) \in S_{xu}(G_i, G_k)$ and $s_{uy}(G_{k+1}, G_j) \in S_{uy}(G_{k+1}, G_j)$:

$$\|s_{xu}(G_i, G_k) \oplus s_{uy}(G_{k+1}, G_j)\| = \|s_{xu}(G_i, G_k)\| + \|s_{uy}(G_{k+1}, G_j)\| - \delta_u$$

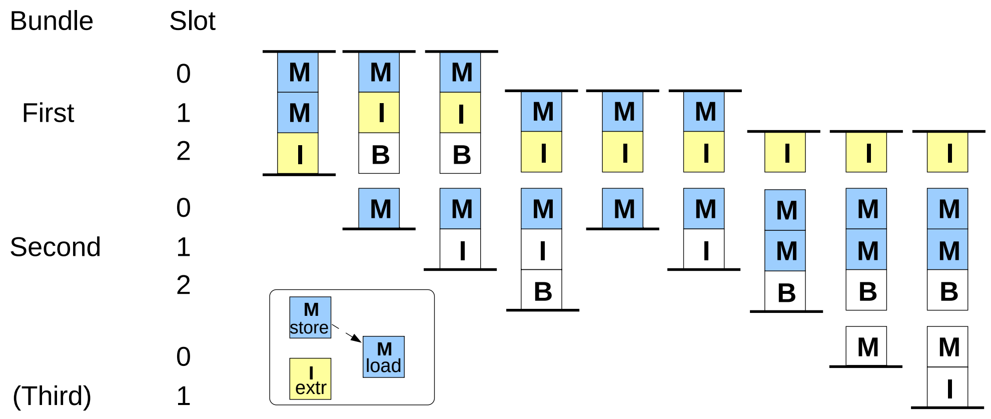


Figure 6.13: Nine possible bundle sequences for an instruction group with three instructions; nops are shown with a white background. The horizontal bars represent the stops.

$$\delta_1 = \delta_2 = 1 \qquad \delta_0 = 0$$

The factor δ_u allows for the possibility that two partially used bundles are merged into one if the concatenation occurs at an intra-bundle stop ($u = 1, 2$).

On the basis of this notation an algorithm is now presented that finds one element of each of the sets $S_{xy}(G_i, G_j)$—referred to as $o_{xy}(G_i, G_j)$—that is *optimal*, i.e., that does not cause a split issue and is of minimal length. The computation of the small sequences $o_{xy}(G_i, G_i)$ ($i = j$) corresponds exactly to the micro-scheduling problem mentioned above. We postpone this problem for the moment and assume that the sequences $o_{xy}(G_i, G_i)$, $\forall i = 1, \ldots, T$, $\forall x, y = 0, \ldots, 2$ are given. These are for each i exactly the nine sequences shown in Fig. 6.13.

We use *dynamic programming* [CLR01] to construct larger optimal sequences from smaller ones *bottom-up*. When constructing an optimal sequence $o_{xy}(G_i, G_j)$, which spans $j - i + 1$ instruction groups, we can assume that the smaller optimal sequences (comprising fewer groups) have already been computed. Candidates for $o_{xy}(G_i, G_i)$ are then all possible concatenations of these smaller optimal sequences that are elements of $S_{xy}(G_i, G_j)$:

$$O_{xy}(G_i, G_j) = \{o_{xu}(G_i, G_k) \oplus o_{uy}(G_{k+1}, G_j) \,|\, i \leq k < j,\ 0 \leq u \leq 2\} \tag{6.6.1}$$

We choose one sequence of minimal size from this set as $o_{xy}(G_i, G_j)$. An implementation has to go through maximally $3(j - i)$ possibilities to find such a sequence. However, often the search can be aborted earlier if a lower bound shows that the obtained sequence must be of minimal size. For example, there must be at least as many slots as instructions in the sequence, thus the following term is a lower bound on $\|o_{xy}(G_i, G_j)\|$:

$$\left\lceil \frac{\sum_{k=i}^{j} |G_k| + x + \begin{cases} 0 & \text{if } y = 0 \\ 3 - y & \text{if } y > 0 \end{cases}}{3} \right\rceil$$

The whole process is repeated iteratively for larger and larger sequences until with $o_{00}(G_1, G_T)$ a bundle sequence for the whole basic block is obtained. Evidently, this sequence is of minimal length since all possibilities of composition were taken into account. The overall worst-case complexity of sequencing is $\sum_{d=2}^{t} 3(d-1) = \mathcal{O}(t^2)$.

It remains to be shown how the base cases $o_{xy}(G_i, G_i)$ can be computed. A micro-scheduling instance is characterized by a start/end slot number pair and up to six instructions with their respective types and intra-group dependences. As mentioned previously, the task to place these instructions into one or two bundles can be regarded as a scheduling problem and also formulated as an ILP [KW01, Win01]. However, due to the limited dispersal window size of the Itanium 2 and the resulting small problem sizes, we can alternatively afford to search these partial sequences via exhaustive enumeration.

To solve a micro-scheduling instance in this way, we simply enumerate all possibilities to map the up to six instructions to the slots of up to two bundles of all possible templates. The remaining unused slots of each possible mapping are then filled with nops of the respective slot types (in Fig. 6.13 depicted by the white slots). After that, the dispersal of the sequence is simulated according to the rules of Sec. 2.11. Only sequences that cause no splits are accepted, and of those only one of minimal length is eventually selected. In Fig. 6.13, for example, only sequences with three bundles exist for the start/end slot combinations 2/1 and 2/2—these two micro-scheduling instances are thus considered infeasible.

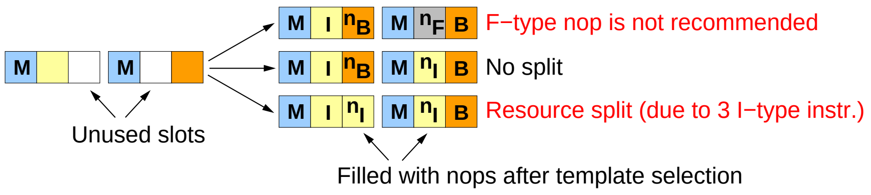

M
M
Unused slots
M I n_B M n_F B
F−type nop is not recommended
M I n_B M n_I B
No split
M I n_I M n_I B
Resource split (due to 3 I−type instr.)
Filled with nops after template selection

Figure 6.14: Unintended consequences of different nop types.

Since nops are dispersed like normal instructions, they are not unproblematic with regard to split issue: For instance, a nop in an I-type slot is regarded as a normal I-type instruction by the dispersal logic and can thus split issue if already two other I-type instructions are in the group (see Fig. 6.14). However, this can be prevented by using a dummy A-type instruction like `add rX=0,rX` in place of an I-type nop (such that register `rX` is not read or written in the group). The advantage of such an artificial "A-type nop" is that it can also be mapped to an M-port as described in Sec. 2.2.3.2. The use of F-type nops is not recommended since these may cause unintended stalls related to writes to the FPSR register [Int04]. B-type nops are unproblematic, but not in combination with the MBB or BBB templates, which always trigger a split issue. Another advantage of B-type nops is that they can possibly be replaced by useful branch predict instructions after bundling (see Sec. 2.1.3)[17].

[17]During sequencing, it is possible to prefer bundle sequences that have B-type nops as *early* as possible, i. e. in one of the first groups. This allows to insert `brp` instructions early in basic blocks which is favorable since the distance between a `brp` and the branch it refers to at the end of the block should be as large as possible. In the

To avoid the expensive enumeration for each given instruction group (and nine start/end slot combinations) during bundling, the results can be *precomputed* for all possible groups and stored in a hash table. For this purpose we need a key to identify a micro-scheduling instance. The following information uniquely describes the relevant characteristics of such an instance:

1. The numbers of instructions of each of the eleven possible instruction types (listed in Sec. 2.2.1) in the group.
2. The start/end slot number pair.
3. The intra-group dependences.

As expressed by the first point, the *numbers* of occurances of different instruction types are already sufficient to characterize a group for bundling—the mnemonics of the individual instructions or their operands are irrelevant. Since these numbers are limited in any feasible instruction group, the first two characteristics can be described using a 32-bit word [Win01, Ste03]. 10291 possible combinations of them were enumerated in the implementation [Ste03]. The last characteristic, however, has the potential to multiply this number by a significant factor: in theory, there can be up to $2^{6 \cdot 5}$ different sets of intra-group dependences. Even if in practice the number of possible and relevant combinations is much lower, it is unrealistic to precompute the results for all of them.

Hence in our solution the third characteristic is ignored when the hash table is created. Instead, the table contains not only one optimal solution for each key, but a list of solutions *for all possible orders* of the instructions in the group. This list has maximally $6! = 720$ entries, but the actual number is lower because not for all orders feasible partial bundle sequences (without split issue) exist [Ste03]. Each entry can be stored in a 32-bit word that encodes a mapping of the maximally six instructions to the slots of up to two bundles ($6 \cdot 3$ bits) plus the templates of these bundles in the solution ($2 \cdot 4$ bits). The list is ordered by increasing lengths so that favorable solutions come first.

During bundling, a sequence $o_{xy}(G_i, G_i)$ is looked up as follows: a key is generated from the instruction types in G_i and x/y. Then the list corresponding to this key is traversed and the first solution is returned whose order does not violate the intra-group dependences of the group. Since the solutions are the results of exhaustive search, it is clear that the partial sequence obtained in this way is optimal and thus the same holds for the solution of the entire bundling phase.

We forgo a complexity analysis of the precomputations here since the implementation shows that they can be performed in a few seconds [Ste03]. Furthermore, the table size of 1.48 MByte in this implementation demonstrates that the approach is viable. Since most instruction groups contain no or not more than one intra-group dependence, usually one of the first entries of the list is returned [Ste03]. Thus only small parts of the table can be expected to be frequently accessed and thus to be present in the caches. In relation to the basic block size, the table lookup requires constant time, hence we can perform optimal bundling in quadratic time.

developed bundler, however, the insertion of these instructions is currently not performed because they are ignored by the branch prediction hardware of the Itanium 2 [Int04].

6.6.1 Bundling Constraints

In the remainder of this section, we will deal with a possibility that we have optimistically ignored in the above description of bundling: it can happen—due to intra-group dependences—that no feasible partial bundle sequence exists for an instruction group at all, even though the latter is feasible as defined in Sec. 5.2.1 (i.e., a mapping of the instructions to the execution units exists). Consider, for example, the following group:

```
A: chk.s r26, .rec_7        //M/I
B: st4 [r20]=r26            //M
C: chk.a.clr r14, .rec_8    //M
D: st4 [r14]=r18            //M
E: ld8.c.clr r8=[r34]       //M
F: add r20=r8,r56           //A
```

The control speculation check `A` must be scheduled before the store `B` (which could otherwise trigger a NaT consumption fault due to `r26`). Furthermore, the data speculation checks `C` and `E` must appear after the stores `B` and `D`, respectively, but before the instructions `D` and `F`, which read `r14` and `r8`, respectively. As a result of these intra-group dependences, the six instructions may only appear in exactly the given order in a bundle pair—but apparently this is impossible (`A` would have to occupy an I-type slot so that according to (2.1.1) none of the four M-type instructions `B`-`E` could be placed in the first bundle, but there is only place for two in the second bundle).

Our definition of "feasible instruction groups" in Sec. 5.2.1 is optimistic in the sense that it regards only the numbers of instructions of different types in the group, but ignores bundling-related issues. We call groups that are feasible according to this definition, but cannot be bundled without split issue due to the structure of their intra-group dependences *structurally infeasible*. They constitute a rarely occurring, but intricate problem that arises from the separation of instruction scheduling and bundling in our approach. In the experiments of Chapter 7, only two such groups (of a similar form as the above example) emerged in the optimal schedule computed for one of the input routines (*qSort3*).

It may be possible to remove structurally infeasible groups in the schedule afterwards by reordering instructions between different groups, but it cannot be relied on that the schedule always permits this postpass remedy. The theoretically ideal solution would be to integrate bundling into the global scheduling phase. In [CLF$^+$03, CLJ$^+$04] it is shown how this integration can be done for a heuristic scheduler that targets the first-generation Itanium. When scheduling instructions into an instruction group, it employs a finite state automaton to keep track of the execution units occupied by them and to ensure that a template assignment exists for it. Each state encodes the currently occupied execution units in the group and is associated with a list of all possible template assignments that comply with this occupation. The scheduling of an instruction into the group triggers a transition between states. Such a transition is only *legal* if at least one template assignment associated to the new state satisfies all intra-group dependences—otherwise the corresponding scheduling decision is rejected. In doing so, the feasibility of bundling is ensured for all instruction groups.

An ILP formulation, however, that incorporates bundling—similar to the one developed for micro-scheduling in [Win01]—would require up to six new variables in place of each x variable alone to model the placement of instructions into different slots. Thus the number of needed x variables would be multiplied and also the number of required precedence constraints (5.3.12) for the intra-group dependences. The experiments indicate that the interdependences between scheduling and bundling are to weak to justify this massive complexity increase.

Instead, our solution is to *prevent* the formation of structurally infeasible groups *in advance* by means of separate *bundling constraints*. For this purpose, we collect for each basic block A and each cycle $t \in G(A)$ all instructions that can be potentially scheduled there in a set $P_{At} \subseteq V$ (typically, $P_{At} := \Theta^{x^{-1}}(A)$). We define a relation $\lhd \subseteq E_D$ on P_{At} such that $m \lhd n$ holds for a pair $m, n \in P_{At}$ if these two instructions *can* appear together in the same group, but then n must appear after m there. To check the former condition, we can employ the minimum distances that will be later computed in Sec. 7.1.2: both instructions can be scheduled into the same group only if $d^{\updownarrow}_{m,n} \leq 0$. The second condition applies if there exists an intra-group DDG edge $(m, n) \in E_D$ (with $w_{mn} = 0$).

The goal is now to find subsets of P_{At} that constitute structurally infeasible instruction groups and to exclude them via additional inequalities. We aim at characterizing these groups as generally as possible. In the above example, the combination that already leads to structural infeasibility (following the above argumentation) is a control speculation check `A` and four dependent M-type instructions `B`-`E` such that `A` $\lhd$ `B`, `A` $\lhd$ `C`, `A` $\lhd$ `D`, and `A` $\lhd$ `E` (`F` is irrelevant). The following bundling constraint then prevents the formation of this group at cycle t in block A:

$$x^{At}_{\mathtt{A}} + x^{At}_{\mathtt{B}} + x^{At}_{\mathtt{C}} + x^{At}_{\mathtt{D}} + x^{At}_{\mathtt{E}} \leq 4$$

We use *patterns* as an intuitive, unified representation of this and further structural infeasibility conditions, as some of them are shown in Fig. 6.15. The intended meaning of these patterns is that each potential instruction group that matches one of them (called a *match*) is structurally infeasible and should be excluded by means of a bundling constraint. A pattern is a graph with nodes $v_1, \ldots, v_k$ such that each node v_i is assigned an instruction type, $R(v_i) \in \vec{\mathcal{R}}$, and a cardinality, $|v_i| \in \mathbb{N}_+$. In Fig. 6.15, only cardinalities greater than one are explicitly shown in parentheses. An instruction group $P' \subseteq P_{At}$ *matches* this pattern if there exists a function $\Phi : P' \rightarrow \{v_1, \ldots, v_k\}$ that maps $|v_i|$ instructions to each node v_i under the following conditions:

1. the instruction types must match the node types and

2. if two nodes v_i and v_j are joined by an edge, then for each pair of instructions m and n mapped to v_i and v_j, respectively, a path from m to n exists in the graph $(P_{At}, \lhd)$.

So to the nodes of the patterns sets of one or more same-type instructions are assigned and the edges represent (chains of) intra-group dependences between the instructions in these sets. The above example group matches Fig. 6.15 (a) by assigning `A` to the upper node and `B`-`E` to the lower node.

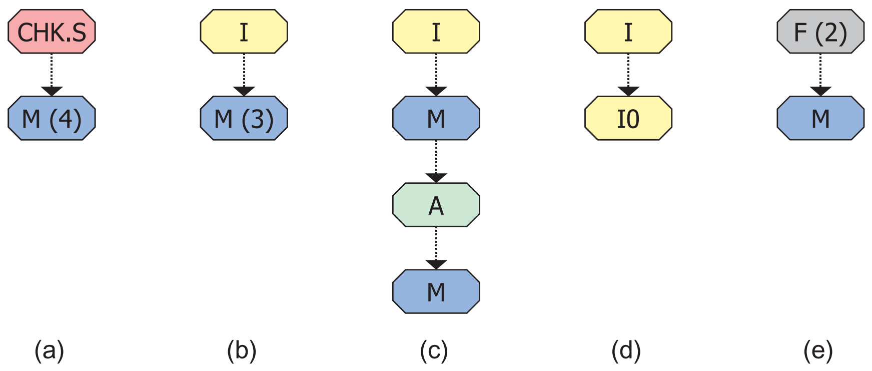


Figure 6.15: Patterns of structurally infeasible instruction groups.

The first three infeasibility patterns in Fig. 6.15 exploit the fact that no M-type instruction can be scheduled after an instruction in an I-type slot in the *first* bundle and that then there is no template for the second bundle that can host more than two M-type instructions ((a), (b)) or the combination MIM/MMM (c). Fig. 6.15 (d) allows for the rule that the first I-slot instruction in a group always occupies I0. The second F-type instruction of the last pattern must be placed in the second bundle with template MFI, MFB or MMF—but then there is no slot for a further M-type instruction after it in the bundle (the M-type instruction could be a `setf` with a WAR (intra-group) dependence on the floating-point instruction).

The bundling constraints for a pattern have the following general form (with $w := \sum_{i=1}^{k} |v_i|$):

$$\sum_{n \in P'} x_n^{At} \leq w - 1 \qquad \forall A \in \mathcal{B},\ \forall t \in G(A),\ \forall \text{ matches } P' \subseteq P_{At} \tag{6.6.2}$$

The number of these constraints is polynomial since $|P'| = w$ must be less than or equal to six. Thus the number of possible matches cannot grow larger than $\binom{|P_{At}|}{6} \leq |P_{At}|^6$. However, this number can still be considerable so that the following reduction is useful: If for a node $|v_i| = c(R(v_i))$ holds, i.e., if its cardinality is equal to the maximal number of instructions of this type in any feasible instruction group, then we can assign not only $|v_i|$, but an unlimited number of instructions to this node, which leads to larger, *fewer* matches. As an example, consider pattern (a) and the previous example with a fifth instruction $\mathtt{F}$ such that $\mathtt{A} \lhd \mathtt{F}$: It is not necessary to instantiate (6.6.2) for all five possible matches $\{\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{D},\mathtt{E}\}$, $\{\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{D},\mathtt{F}\}$, $\{\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{E},\mathtt{F}\}$, $\{\mathtt{A},\mathtt{B},\mathtt{D},\mathtt{E},\mathtt{F}\}$, and $\{\mathtt{A},\mathtt{C},\mathtt{D},\mathtt{E},\mathtt{F}\}$ such that $\mathtt{A}$ matches the first node of the pattern and the other four instructions the second node; instead, one constraint for $P' = \{\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{D},\mathtt{E},\mathtt{F}\}$ is sufficient. For nodes without this property, as those in pattern (b), however, this simplification is not allowed since the bundling constraints would then become too tight and could exclude feasible groups.

To generate all bundling constraints of a pattern, it is necessary to enumerate all matches of it in the graph $(P_{At}, \lhd)$. For subgraph isomorphism in general, no better algorithm than an exhaustive enumeration and checking of all potential matches is known [Epp95] ($\mathcal{O}\left(|P_{At}|^6\right)$ comparisons here). For patterns of a simple tree-like or even linear structure as those in Fig. 6.15, however, it is apparent that a recursive graph search can find all matches with much fewer comparisons than this worst case bound.

The five patterns of Fig. 6.15 represent only a selection of possible bundling constraints; they are not necessarily complete. Further conditions of split issue could occur, for example, when intra-group dependences force A-type instructions to occupy M-type slots before other M-type instructions—dealing with these cases would require knowledge of the undocumented part of the dispersal rules (see Sec. 2.10, [Int04]). To find all necessary bundling constraints and to prove that they are sufficient to exclude all possible structurally infeasible groups, a complete, formal description of the dispersal logic would be needed.

The bundling constraints are an example of how we favor an approximate, lean approach over a precise one in order to avoid a prohibitive complexity increase as it would be the consequence of integrating bundling into the main scheduling model. Despite its high worst case complexity, it turned out to be practicable during the experiments (see Sec. 7.3.1). If in the course of broader experiments further structurally infeasible groups appear, they would imply further classes of bundling constraints. These constraints, possibly in combination with postprocessing, appear as the most efficient way to cope with the phase coupling between scheduling and bundling.

6.7 Future Work

In this section, we consider possible further improvements and extensions of the developed formulations. One useful enhancement has already been indicated in Sec. 6.1.2: There it has been observed that the collapsing of blocks can lead to branches whose conditions are not available in predicate registers. It would be beneficial to compute these predicates "on demand" in the schedule: For each of them that is possibly needed, a set of additional compares could be derived and included in the ILP that computes it. This set should only then be included in the schedule if a branch with this condition is actually required, which can be modeled similarly as the mutually exclusive sets of instructions.

In general, we can subsume such sets of instructions, which should occur in the schedule only under a certain condition (that is modeled in the ILP), under the notion "*conditional sets of instructions*". They constitute a versatile basis for the integration of complex code generation decisions into the model. The preceding sections have shown this for speculation decisions, including non-exclusive use forking, the speculation scheme sketched in Sec. 6.2.2 whose potential has not yet been fully investigated. As a result of the intensive use of these conditional sets, instructions from them can account for a large fraction of all instructions modeled in an ILP, possibly even the majority.

Thus, a future, extended analysis of the efficiency of the model should focus on these sets and could even incorporate them—right from the beginning—in the development of the basic ILP model and the accompanying proofs of integrality properties in Chapter 5. This promises further efficiency improvements: In fact, later in Chapter 7, we will describe how the precedence constraints can be tightened by exploiting the mutual exclusiveness of instructions.

The analysis of partial-ready code motion has revealed further opportunities for fundamental enhancements of the model: it has led to a more differentiated understanding of data dependences, namely that true dependences constitute the requirement to schedule instructions (in a certain range) and false dependences the exclusion. This distinction is currently not reflected by the model, with the consequence that PR code motion must be restricted in the presence of WAR dependences (see Sec. 6.3). This could be avoided by handling these dependences differently, using the following new variables:

$z_n^{\uparrow A} = 0 \quad \Rightarrow$ No copy of instruction n is scheduled on any program path through $s(n)$ before A.

The condition expressed by these variables is contrary to that modeled by the a variables (yet not directly the logical complement). Their semantics could be implemented very similarly, using constraints like Equ. (6.3.1); the additional complexity is comparable to that of the a variables and thus relatively low. It is apparent how they can be employed to model WAR dependences in compliance with Def. 6.3.6. It is even conceivable to allow for the refined data dependence preservation—as for the conditional sets—directly in the development of the basic model, so that the z variables are not integrated afterwards, but introduced together with the a variables. This could help keep the integrality result from Sec. 5.1.2.

The developed ILP model can be transformed to support *software pipelining* in a straightforward way. Details on the reformulation are provided in [Win01]. The unfolded software pipeline is scheduled as a single basic block; Special modulo resource constraints model the "folding" of this pipeline within the kernel loop (in order to execute its different stages in parallel). The kernel length (the Initiation Interval II) is assumed constant in the ILP, which reduces its complexity considerably. Thus possibly multiple ILPs with different values of II have to be solved in order to find a minimal-length kernel by means of binary or linear search.

As the high parallelism of the kernel loop entails a high number of simultaneously live values, modeling the allocation of the rotating registers, possibly with spilling, could be considered. An earlier study has shown that heuristic modulo scheduling compares very well with an ILP-based exact approach [RGSL96]. However, the results of the latter do not guarantee register optimality for many loops. The outcome may be different on an EPIC architecture with architectural support for software pipelining.

This chapter closes with an outline of how one of the presumably most complex, but also most important extensions could modeled, namely the *inter-block propagation of long latencies*. Currently, stalls caused by these latencies are not measured and allowed for at all so that the model underestimates their impact. How large the inter-block stall triggered by a use-definition pair is depends on which control flow path between the two instructions is taken at runtime: this path decides how many cycles are spent between the execution of the two instructions and thus how much of the latency can be *covered* (by these cycles).

Fig. 6.16 (a) gives an example of this: the (minimum) latency of the floating-point load `ldfs` there is 6 cycles. Along the path A-B-D, four of these cycles are covered by the execution of other instructions (not explicitly shown), so that a one-cycle stall in block D at the use `fma` results. Along the path A-C-D, this stall is one cycle longer since one cycle less is covered. When the stalls are added to the global schedule length (3.3.8), they should be *weighted by the respective path frequencies*, which yields in this example $1 \cdot 0.6 + 2 \cdot 0.4 = 1.4$.

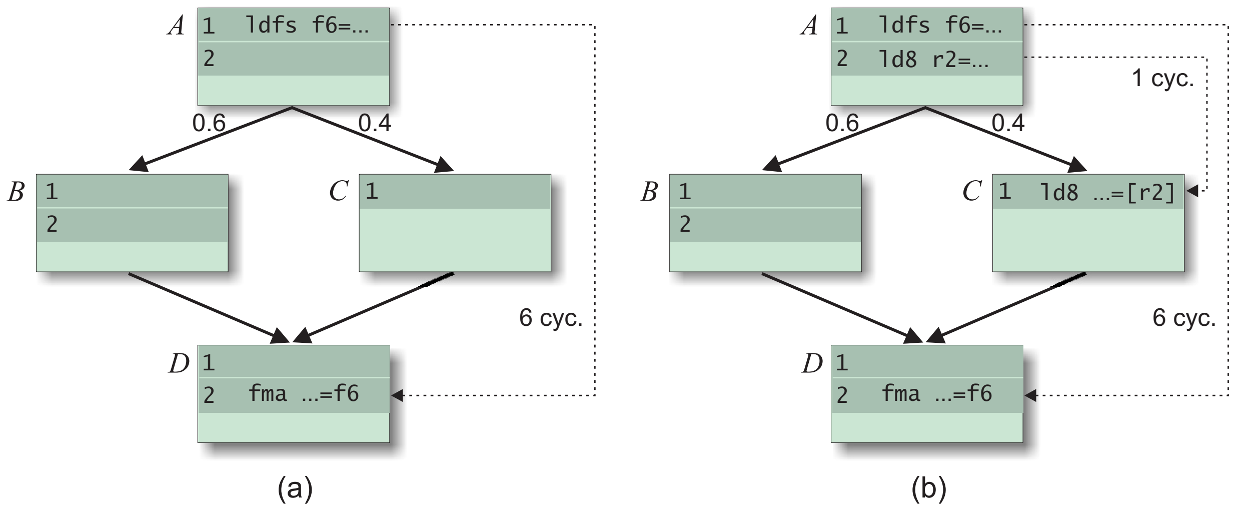


Figure 6.16: Case study of inter-block latency propagation.

Thus, a precise model of inter-block stalls must determine them and take them into account on a *per-path basis*. This requires for each program path $P \in \mathcal{C}$ an execution frequency f_P by which the measured stall cycles along this path are weighted. These values can be taken from path profiles (see Sec. 3.1), or, if these are not available, approximated from node or edge profiles [SS02]. If they are included as *constants* in the ILP, however, then the worst-case complexity of the latter becomes exponential (which is the maximal number of program paths through the scheduling region). It is unclear whether this exponential complexity can be avoided, in other words, whether a precise modeling of inter-block stalls exists that is polynomial sized.

Since the path frequencies are often only approximations obtained from edge or node profiles, anyway, the presumed high complexity of a precise, path-based model does not seem reasonable. Instead, we follow here an approximate approach that considers latency propagation not along individual paths, but along *bundles of paths that traverse a pair of blocks*. The formulation outlined in the following employs for each pair of blocks $A, B \in \mathcal{B}$ such that $A \prec B$ the following integer variable:

$s_{A \dashrightarrow B}$ = The stall cycles in B when a program path from $\mathcal{C}(A) \cap \mathcal{C}(B)$ is executed.

Each of these variables, $s_{A \dashrightarrow B}$, is added to the objective function and there weighted by $f_{A,B} := \sum_{P \in \mathcal{C}(A) \cap \mathcal{C}(B)} f_P$, the aggregate frequency of all program paths that pass through both A and B (which can be approximated from an edge profile):

$$\sum_{\substack{A,B \in \mathcal{B} \\ A \prec B}} f_{A,B} \cdot s_{A \dashrightarrow B}$$

The value of this sum is also referred to as the *total stall value*.

Stalls are the result of latencies that are *propagated* between blocks. More precisely, we say that a certain latency is propagated from block A to block B (or *between* A and B) if the direct execution of B after A (without other intermediary blocks) would cause a stall of this duration. In Fig. 6.16, for example, a three-cycle latency is propagated from A to D. A separate, further class of integer variables[18] is used to model the propagation between block pairs:

$$l^{C}_{A\dashrightarrow B} \quad = \quad \text{The latency propagated from block } A \text{ to } B \text{ at the exit of block } C.$$

These variables are instantiated for all $A, B, C \in \mathcal{B}$ such that $A \preceq C \prec B$. The differentiation of the latencies according to the additional block C models that they *decrease* monotonously during the propagation along control flow paths (as described below). The precise semantics of these variables is as follows: if B would be executed directly after C on a path from $\mathcal{C}(A) \cap \mathcal{C}(B)$, then a stall of $l^{C}_{A\dashrightarrow B}$ cycles would occur.

As embodied by the defined variables, the latency propagation is modeled between *block pairs*—although the latencies are triggered by pairs of *instructions*. More precisely, the latency propagated from a block E to a successor block F—the value of $l^{E}_{E\dashrightarrow F}$—must be greater or equal to the maximum latency propagated from a copy of an instruction m scheduled in E to a copy of an instruction n scheduled in F. If $(m, n) \in E_D$ and such copies of m and n are scheduled in cycles $s_{m,E}$ and $s_{n,F}$ in E and F, respectively, then the latency propagated between them equals the total latency w_{mn} minus the cycles that are covered by other instructions in E (after $s_{m,E}$) and F (before $s_{n,F}$):

$$l^{E}_{E\dashrightarrow F} \geq w_{mn} - (T_E - s_{m,E}) - s_{n,F}$$

These constraints are only schematic; their final form in the ILP model is different since there the variables $s_{m,E}$ and $s_{n,F}$ are not directly available as integers (we do not expand on possible implementations and their efficiencies here). The purpose of the constraints is to *initiate* inter-block latencies; the other remaining constraint classes, to be presented in the following, *propagate* them downwards in the BBG and *transform* them eventually into stalls. When a latency from a block E to a successor F is propagated along a control flow path (from $\mathcal{C}(E) \cap \mathcal{C}(F)$), it is diminished (covered) by

- the schedule lengths of the encountered blocks,
- the stall cycles incurred in these block on the same control flow path, and
- the latency cycles that are propagated from other blocks on this path to F—if these cycles were not subtracted, they would be taken into account twice.

More concretely, when a latency from a block E to a successor F is propagated through a block H on this control flow path, then it is diminished by

- the schedule length T_H,

[18]It is also thinkable to represent latencies and stalls with binary variables (like the block lengths) since they are always integral. This could enable a tighter ILP model, but it has the drawback that the complexity of the formulation then is no longer independent of the latency values, but grows with them.

- the stall cycles $s_{E\dashrightarrow H}$, and
- the latency cycles $l^H_{H\dashrightarrow F}$ (since the stall variable $s_{H\dashrightarrow F}$ already takes account of these cycles).

This is implemented by the following inequalities:

$$l^G_{E\dashrightarrow F} - T_H - s_{E\dashrightarrow H} - l^H_{H\dashrightarrow F} \leq l^H_{E\dashrightarrow F} \qquad \forall E,F,G \in \mathcal{B}: \begin{array}{c} E \preceq G \wedge \\ (G,H) \in E_B \wedge H \prec F \end{array} \tag{6.7.1}$$

The latency that eventually arrives at F determines the stall that occurs there:

$$l^G_{E\dashrightarrow F} \leq s_{E\dashrightarrow F} \qquad \forall E,G,F \in \mathcal{B}:\ E \preceq G \wedge (G,F) \in E_B$$

In the example of Fig. 6.16 (a), a three-cycle latency is propagated from block A to D ($l^A_{A\dashrightarrow D} = 3$). Along the paths through B and C, the instances of (6.7.1) with $H := B$ and $H := C$, respectively, look as follows:

$$\begin{aligned} 1 = \underbrace{l^A_{A\dashrightarrow D}}_{=3} - \underbrace{T_B}_{=2} - \underbrace{s_{A\dashrightarrow B}}_{=0} &\leq l^B_{A\dashrightarrow D} + l^B_{B\dashrightarrow D} \\ 2 = \underbrace{l^A_{A\dashrightarrow D}}_{=3} - \underbrace{T_C}_{=1} - \underbrace{s_{A\dashrightarrow C}}_{=0} &\leq l^C_{A\dashrightarrow D} + l^C_{C\dashrightarrow D} \end{aligned}$$

This allows the solution $l^B_{A\dashrightarrow D} = l^C_{A\dashrightarrow D} = l^C_{C\dashrightarrow D} = 1$ and $l^B_{B\dashrightarrow D} = 0$, which yields a total stall value of $s_{A\dashrightarrow D} \cdot 1 + s_{C\dashrightarrow D} \cdot 0.4 = 1.4$ in the objective function. Remarkably, in this solution one cycle of the latency propagated from A to D at block C is effectively transferred to the variable $l^C_{C\dashrightarrow D}$ although no instructions in C and D trigger this latency. If this split-up of the latency would not take place, the value of $s_{A\dashrightarrow D}$ had to be two and a total stall of the same value would be measured. This demonstrates the ability of the formulation to *balance* propagated latencies between smaller and larger path bundles as a means of covering and taking them into account on a finer grain, leading to smaller, more precise total stall values.

In Fig. 6.16 (b), we assume that a further one-cycle latency is propagated from A to C (the load$\rightarrow$load address latency is two cycles, see Sec. 2.2.3.3). The resulting additional stall at C does not increase the total stall value since it overlaps with the latency propagated from A to D through this block, decreasing it by the same value. The feasible solution $l^A_{A\dashrightarrow C} = l^B_{A\dashrightarrow D} = l^C_{A\dashrightarrow D} = 1$ and $l^B_{B\dashrightarrow D} = l^C_{C\dashrightarrow D} = 0$ with the same total stall value $s_{A\dashrightarrow D} \cdot 1 + s_{A\dashrightarrow C} \cdot 0.4 = 1.4$ demonstrates this.

In both examples, the measured total stall values are equal to that of the precise formulation—however, this does not hold in general. The formulation is still an approximation that assigns uniform latency and stall values to path *bundles* and not to individual paths. However, if the precise latency and stall distribution can be expressed as a superposition of these bundle-related latencies and stalls (which is presumably often the case), then both models measure the same total stall value. Otherwise, the approximation may overestimate the precise value.

In the current form, the extension supposes that no *intra*-block stalls occur since the precedence constraints take long latencies locally into account so that two dependent instructions m

and n are always scheduled at least cycles w_{mn} apart in a basic block. However, latencies inside a block can vary in the presence of predication: Consider two instructions m and n with $w_{mn} > 1$ that are scheduled via predicated code motion in the same basic block. Then the long latency w_{mn} is only then propagated from m to n (and can induce a stall) if both instructions are *predicated on*—otherwise it is effectively equal to one (see Sec. 2.2.3.3). The current formulation does not allow for this; it considers intra-block latencies static and consequently penalizes predicated code motion of long-latency instructions.

A possible remedy for this is extensive and therefore here only coarsely sketched: A first, inevitable step is to switch from the current *time-indexed* to a *group-indexed* formulation. That is, the index t of the x variables is redefined to refer to *instruction groups* instead of *cycles*. The difference between these two measures constitutes the *stall cycles*. These are modeled separately via new variables: For each $t \in \{1, \ldots, \mathbb{G}_A - 1\}$, an integer variable is added that measures the stall between the execution of the groups t and $t + 1$ in block A. In the objective function, these stall variables are weighted by the block's execution frequency, f_A.

To allow for varying stalls due to predication, we introduce not only one stall variable per group t, but several classes of them. If instruction pairs like m and n from above are scheduled in A via predicated code motion, then they define a class of stall variables of their own that is weighted in the objective function by the frequency that A is executed *and* both instructions are predicated on. Constraints ensure that intra-block latencies and stalls can be balanced between different classes, similarly to the inter-block formulation.

Many details still have to be clarified on the way to an ILP model of long latencies in global scheduling. Nevertheless, the presented draft indicates that a close approximation is—at the price of $\mathcal{O}\left(\mathbb{G} \cdot |\mathcal{B}|^2\right)$ additional variables—feasible .

Chapter 7

Experimental Evaluation

The development of the ILP model with its extensions was closely accompanied by a continuous experimental evaluation. This ensured the feasibility of the developed formulations as well as their relevance with respect to performance. During the experiments, Intel's compiler for the Itanium, `icc`, was taken as a reference: the goal was to improve performance relative to this state-of-the-art product compiler [DKK+99, BCC+00]. Since a direct integration of the ILP scheduler into `icc` was not possible, a postpass approach was used instead. This limits the scope of the experiments, but it still allows a direct comparison with Intel's code generator (which employs wavefront scheduling from Sec. 3.3.1), giving an idea of its perfomance potential. During the comparison, however, it should always be kept in mind that both approaches play in different leagues regarding the computation times.

7.1 Implementation

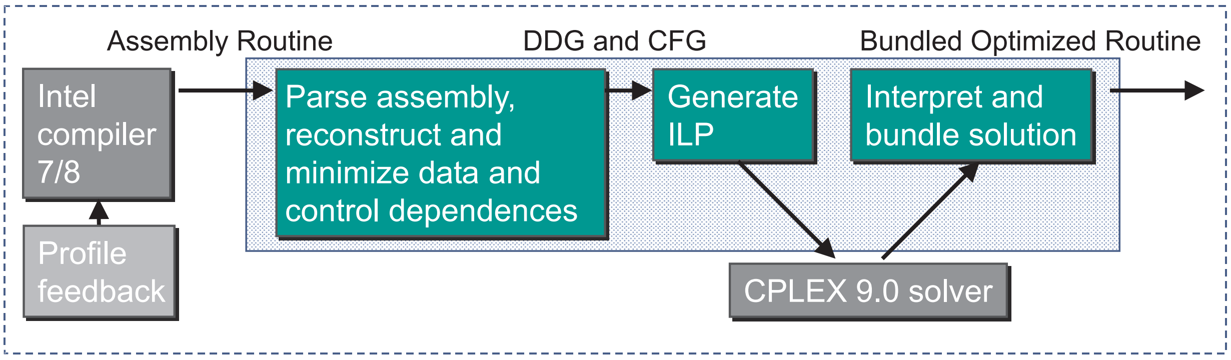

Assembly Routine
DDG and CFG
Bundled Optimized Routine
Intel compiler 7/8
Profile feedback
Parse assembly, reconstruct and minimize data and control dependences
Generate ILP
Interpret and bundle solution
CPLEX 9.0 solver

Figure 7.1: Overview of the implementation.

Fig. 7.1 provides an overview of the implemented ILP optimizer: It reads directly assembly routines produced by `icc` (using the `-S` flag) and reconstructs control flow and data dependences. It also reads the execution frequency estimates of the basic blocks that are annotated by

Intel's compiler in the assembly code. This information is available from earlier profiling runs.

After the generation of the ILP, the tool invokes the solver and waits until the latter terminates. Once the solution is available, it constructs the schedule from it and activates the bundler. The whole process is fully automated, only in some cases manual adjustments are necessary. This is detailed in the following more comprehensive description.

7.1.1 Parsing and Precomputations

The parser is designed to process entire procedures. It recognizes the types and operands of the instructions, reconstructs the basic block graph, and computes the dominance and postdominance relationships using a standard control flow analysis [Muc97]. The dominance relation is then used to identify backedges and loops. Software-pipelined loops are also recognized and merged into single, non-moveable instructions that summarize the data flow to and from them (similarly as described for calls in Sec. 6.1.1).

The qualifying predicates of the conditional branches at the ends of the blocks are read out and associated to the respective BBG edges. If there is an `alloc` at the beginning of the procedure, then the sizes of different areas of the register stack frame are extracted from this instruction; they are needed to determine the input registers of calls.

Then the global data dependence graph is constructed from the control flow graph. This is done using a simple kind of forward data flow analysis. It computes at each node of the control flow graph a table that records two sets for each register: these contain those instructions that could have written and read the current value of the register, respectively, at an earlier program point.

At each node (instruction) of the control flow graph, the sets of the instruction's source and destination registers are looked up in the table. For each source register, RAW dependence edges on all instructions from the first set are added. In a similar way, WAW and WAR dependences are detected. An encountered definition of the register clears both sets (cf. Def. 3.2.6)—afterwards, the definition is added to first set. The clearing reflects the overwriting of the register value by the definition—it may only be done for *unpredicated* definitions. Multiple predicated definitions can and should only then clear the sets if under any possible predicate register assignment one of them is active. This condition is determined by recording complementary predicate registers, as outlined in the next paragraph. At a join, the sets are merged. The forward analysis moves once along each backedge to detect the backward dependences for cyclic code motion.

Whenever the analysis encounters a compare that writes two predicate registers with complementary boolean values, then these two predicate registers are recorded as complementary in the table. Then no dependences between instructions are registered that are guarded by these mutually exclusive predicate registers. Multiple dependences of different types between the same pair of instructions are summarized into one single DDG edge with the maximum latency of those dependences. Thus in the implementation, a DDG edge can be composed of multiple *subedges* that represent distinct RAW/WAR/WAW register or memory dependences between the same pair of instructions. Precedence constraints are only generated for the summarized edges—this avoids separate constraints for each of the subedges, which would be redundant.

An inherent drawback of the postpass approach is that no information about memory disambiguation is available. A comprehensive alias analysis, as performed by Intel's compiler [GLS01], is not possible at assembly level. Hence all memory dependences must be reconstructed conservatively. For this purpose, all loads and stores are regarded as references to a single, special "memory register" in the above analysis. The conservative reconstruction puts the postpass optimizer at a disadvantage compared to the compiler since it means less scheduling freedom for memory instructions in the presence of stores.

The tool then undoes all uses of control and data speculation (with manual interaction for some instances). After that, it performs register renaming to remove as many false dependences as possible, as required in Sec. 6.2.1. In the process of renaming, also reaching definitions and concurrent definitions are recorded. The tool does not undo predication where it is used by Intel's compiler. In the input routines this feature is used rarely and conservatively so that we see no benefit in reversing these decisions. The optimizer also does not try to detect and undo speculative code motion, or to merge multiple compensation copies stemming from the same instruction.

Next, the speculation possibilities are computed as described in Sec. 6.2. When adding data speculation possibilities, we encounter the difficulty that no aliasing probabilities are available and that at assembly level certainly not enough information is present to estimate them. Assuming that they are always zero for any store-load pair could advantage the postpass optimizer over the compiler—which does not apply this feature in such an opportunistic way—if this assumption is confirmed at runtime. Otherwise, however, data speculation would incur considerable recovery penalties. Thus our policy towards data speculation is more restricted: we include such possibilities only for store-load pairs that are independent under the ANSI C aliasing rules[1] [ANS89]. In these cases it is assumed that aliasing is unlikely so that the cost of recovering does not need to be taken into account.

After that, the candidate block ranges are determined as described in the previous chapters, together with possibilities for partial-ready and cyclic code motion; `alloc` instructions, calls, and moves to and from the predicate registers are excluded from global code motion since the potential scheduling scope of these rare instructions is highly restricted anyway. Then eventually a choice has to be made with regard to the values $\mathbb{G}_A$—the numbers of reserved cycles of the basic blocks (see the discussion on page 135). In the implementation, $\mathbb{G}_A$ is pragmatically set to the length of A in the input schedule plus one cycle as "headroom". During the experiments, manual additions for individual blocks are performed in two cases (see Sec. 7.2). Increasing the headroom *generally* to two cycles, however, yields more than doubled solution times on difficult problems (without further improvements of the schedule lengths)—this demonstrates how sensitive the ILP complexity is with regard to this value. Thus it must be chosen cautiously; it is the only factor that may require manual adjustments to balance the two goals of optimal solutions on the one hand, and acceptable solution times on the other hand.

[1]These rules forbid access to the same data through pointers that have different types.

7.1.2 Optimizations

The tool performs several optimizations to minimize the data dependence graph and the candidate block ranges. The removal of redundant DDG edges—including backedges—according to Def. 3.2.7 is important to prevent redundant precedence constraints. Subsequent optimization passes then remove impossible candidate blocks as expressed by Equ. (5.3.2), (5.3.3), and (5.3.4) in Sec. 5.3. For dependence edges from E_D^{PR}, Equ. (6.3.3) and (6.3.4) must be applied instead of Equ. (5.3.2). This minimization of the candidate block ranges—which is also extended to cyclic candidate blocks—is essential to reducing the number of x variables.

We go even further and eliminate these variables not only on a per-block basis, but also on a *per-cycle basis*. For this purpose, we determine for each pair of instructions the *minimum distance* that must separate the two (due to data dependences) if they are scheduled in the same basic block. For two instructions m and n, a value for this distance $d_{m,n}^{\updownarrow A}$ in a basic block $A \in \Theta^x(m) \cap \Theta^x(n)$ can be found as follows: Let $V_C \subseteq V$ denote all instructions with their source block on the control flow path $C \in \mathcal{C}$. If we initially ignore cyclic and PR code motion, then for any $C \in \mathcal{C}(A) \cap \mathcal{C}(s(m)) \cap \mathcal{C}(s(n))$ the length of the longest path from m to n in $G_D[V_C]$ is a lower bound on $d_{m,n}^{\updownarrow A}$ since all instructions on this path must be scheduled between m and n in A.[2] Hence $d_{m,n}^{\updownarrow A}$ can be chosen as the maximum of these lengths over all control flow paths $C \in \mathcal{C}(A) \cap \mathcal{C}(s(m)) \cap \mathcal{C}(s(n))$.

In the implementation, however, we choose the *minimum* of these lengths over all $C \in \mathcal{C}(s(m)) \cap \mathcal{C}(s(n))$. The resulting bound, denoted by $d_{m,n}^{\updownarrow}$, may be lower, but is also more general: it holds independently of the destination block and is compatible with PR code motion, which allows that scheduled copies respect only the dependences from a single control flow path. If for one of the control flow paths no path in $G_D[V_C]$ from m and n exists, then $d_{m,n}^{\updownarrow}$ is set to -1 to express that no minimum distance exists.

These bounds are now used to remove x variables as follows: If $d_{m,n}^{\updownarrow}$ is greater than zero for two instructions m and n, then we can eliminate for each block $B \in \Theta^x(m) \cap \Theta^x(n)$ such that no predecessor of B is element of $\Theta^x(m)$ all x_n^{Bt} variables of the first $d_{m,n}^{\updownarrow}$ cycles (i.e., substitute them by zero). We can do this because if one of these variables is equal to one, then a copy of m must be scheduled $d_{m,n}^{\updownarrow}$ cycles earlier in the same block or in a predecessor block, which is not possible since $t \leq d_{m,n}^{\updownarrow}$ and no predecessor exists in $\Theta^x(m)$. If n can be subject to partial-ready code motion ($(m, n) \in E_D^{PR}$), however, then we can perform this removal only if $s(m)$ is a predecessor of B or if it postdominates the latter *and* no other block in $\mathsf{PR}^{\mathcal{B}}(n)$ is a successor of B. Then each path through B is $s(m)$-defined so that the above argumentation applies.

Analogously, we can remove the last $d_{m,n}^{\updownarrow}$ x_m^B variables of a block $B \in \Theta^x(m) \cap \Theta^x(n)$ such that no successor of B is element of $\Theta^x(n)$. In doing so, we effectively derive from the DDG global ASAP/ALAP[3] ranges of cycles for each instruction. This precise analysis reduces the number of x variables of the ILPs by one fifth on the average.

[2]If speculation (Sec. 6.2) can reduce this path length, then this possibility must be taken into account. Thus, when the longest path is searched in the DDG subgraph, it must be assumed that all speculation possibilities are utilized.

[3]As soon as possible/as late as possible

7.1.3 ILP Generation and Solving

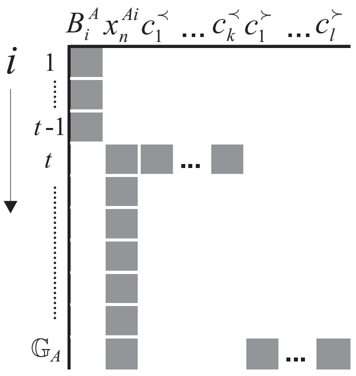
During the generation of the ILPs, we exploit several opportunities to tighten some constraint classes of the model further. Two of them are detailed in what follows. To discover opportunities for tightening, it is helpful to take a closer look at the solutions of the LP-relaxations of real problem instances and especially at the schedules implied by these solutions. Fig. 7.2 (a) depicts a small excerpt from such a *relaxed schedule*, a schedule obtained from interpreting the optimal solution of the *LP-relaxation*. In this real-valued solution, non-integral values of the variables x_n^{At} represent only *partly made* scheduling decisions. As shown in the figure, instruction that are "partly scheduled" in this way are also listed in the relaxed schedule, together with the value of the x variable that represents the scheduling decision in parentheses; this value from the interval $]0, 1[$ can be regarded as a *weight* or as a *bias*.

```
1       ld1.s r23=[r11]  (m')  (0.5)
        sxt r3=r23       (n)   (0.5)

2
   (p8) ld1 r23=[r11]    (m)   (0.5)

3
        sxt r3=r23       (n)   (0.5)
```

(a)


B_i^A x_n^{Ai} $c_1^{\prec}$ ... $c_k^{\prec}$ $c_1^{\succ}$ ... $c_l^{\succ}$
i
1
t-1
t
...
$\mathbb{G}_A$

(b)

Figure 7.2: Illustration of opportunities to tighten the model further.

Fig. 7.2 (a) shows three cycles of a possible relaxed schedule with three instructions: the "`sxt r3=r23`" (n) is dependent on a load "`(p8) ld1 r23=[r11]`" (m), of which also a speculative version "`ld1.s r23=[r11]`" (m') exists. This small example represents an undecided state with respect to control speculation (which is common in relaxed solutions): in this case it is assumed that the control speculation decision variable S_m has value 0.5. Therefore each of the two mutually exclusive instructions m and m' is scheduled only with weight 0.5 in this block A (and we assume that none of the three instructions is scheduled (partly) elsewhere, so that $a_m^{\uparrow A} = a_{m'}^{\uparrow A} = a_n^{\uparrow A} = 0$ and $a_m^{\uparrow B} = a_{m'}^{\uparrow B} = 0.5$ for a direct successor $B \in \Theta^a(m) \cap \Theta^a(m')$ of A).

It can be observed that in this relaxed schedule the copy of n in cycle one violates the data dependences on the copies of both m and m'. Nevertheless, the schedule represents a feasible real-valued solution; it fulfills the following instances of the precedence constraints (5.3.12) for $t = 1$ with equality, but does not violate them (which is possible since the copies are only partly scheduled, i.e., $x_n^{A1} = x_{m'}^{A1} = x_m^{A2} = 0.5$):

$$a_n^{\uparrow A} + x_n^{A1} + (x_m^{A1} + x_m^{A2} + x_m^{A3}) + (1 - a_m^{\uparrow B}) \leq 1 + S_m \quad (7.1.1)$$
$$a_n^{\uparrow A} + x_n^{A1} + (x_{m'}^{A1} + x_{m'}^{A2} + x_{m'}^{A3}) + (1 - a_{m'}^{\uparrow B}) \leq 1 + (1 - S_m) \quad (7.1.2)$$

Now we can exploit the fact that n is dependent—with the same latency—on two instructions that are mutually exclusive, and merge both instances into the following single inequality:

$$a_n^{\uparrow A} + x_n^{A1} + (x_{m'}^{A1} + x_{m'}^{A2} + x_{m'}^{A3}) + (x_m^{A1} + x_m^{A2} + x_m^{A3}) + (1 - a_m^{\uparrow B} - a_{m'}^{\uparrow B}) \leq 1$$

This constraint equals (7.1.1) and (7.1.2) if $S_m = 0$ (then all variables of instruction m' are equal to zero) and $S_m = 1$ (then all variables of instruction m are equal to zero), respectively. Thus the merged constraint does not change the set of integral solutions, but it is *tighter* and excludes the above example (for which its left-hand side has value 1.5). In the implementation, these combined precedence constraints are employed for all dependences on speculation candidates.

Above, we have tightened a constraint class by exploiting the mutual exclusiveness of variables due to speculation choices. In general, two variables are said to be mutually exclusive in the ILP if no integer feasible solution exists in which both have value one. Inequalities of the form $X_1 + \ldots + X_k \leq 1$, where $X_1, \ldots, X_k$ are binary variables, can be strengthened by adding a further binary variable X_{k+1} to the left-hand side that is mutually exclusive to each of the other variables. This does not affect the set of integer feasible solutions (since all other variables must be zero if $X_{k+1} = 1$), but it tightens the constraint by *lifting* it to a higher dimension [NW88].

In Chapter 5 we have shown for most constraint classes that they represent integral facets of subpolytopes so that they cannot be further strengthened with respect to these subpolytopes. However, this may be different for the entire polytope of the model, which is the intersection of all subpolytopes. We focus here on one class for which we have no integrality result at all (due to the $\mathcal{NP}$-hardness of related subproblems): the block length constraints (5.3.14). An instance of these constraints, generated for a block $A \in \mathcal{B}$, an instruction $n \in \Theta^{x^{-1}}(A)$, and a cycle $t \in G(A)$ has the following form:

$$B_1^A + \ldots + B_{t-1}^A + x_n^{At} + \ldots + x_n^{A\mathbb{G}_A} \leq 1 \quad (7.1.3)$$

Fig. 7.2 (b) illustrates the structure of this inequality graphically: the boxes show which variable of each cycle it comprises (initially, the columns marked with $c^{\prec}/c^{\succ}$ should be ignored). No further variable B_i^A can be added to the left-hand side since such a variable for an $i \geq t$ would not be mutually exclusive to x_n^{Ai}. A variable x_m^{Ai} of another instruction m, however, is mutually exclusive to all B variables of the constraint if $i \geq t$. It remains to be determined under which circumstances it is also mutually exclusive to all the variables $x_n^{At}, \ldots, x_n^{A\mathbb{G}_A}$, as required.

For this purpose, we utilize the previously computed minimum distances between instructions that are scheduled in the same basic block: if $d_{m,n}^{\updownarrow} > 0$, then $x_m^{A\mathbb{G}_A}$ is mutually exclusive to all the variables $x_n^{At}, \ldots, x_n^{A\mathbb{G}_A}$—this is because if m is scheduled at cycle $\mathbb{G}_A$, then n must not be scheduled before $\mathbb{G}_A + d_{m,n}^{\updownarrow} > \mathbb{G}_A$. Thus $x_m^{A\mathbb{G}_A}$ can be added to the left-hand side of (7.1.3).

As depicted in Fig. 7.2, we add not only the x variable of one instruction with this property, but of as many as possible. When adding the variables $x_{c_1^{\succ}}^{A\mathbb{G}_A}, \ldots, x_{c_l^{\succ}}^{A\mathbb{G}_A}$ of several instructions

$c_1^{\succ}, \ldots, c_l^{\succ} \in V$ such that $\forall i : d^{\updownarrow}_{c_i^{\succ}, n} > 0$, however, we must make sure that these variables *itself* are mutually exclusive to each other, too. This condition can—based on the minimum distances—be expressed as follows:

$$\forall i, j : \left(d^{\updownarrow}_{c_i^{\succ}, c_j^{\succ}} > 0 \right) \vee \left(d^{\updownarrow}_{c_j^{\succ}, c_i^{\succ}} > 0 \right)$$

During the generation of the block length constraints, such variables are added until no further instruction exists in $\Theta^{x^{-1}}(A)$ that satisfies the above conditions. In an analogous way, variables $x^{At}_{c_1^{\prec}}, \ldots, x^{At}_{c_k^{\prec}}$ are added such that $\forall i : d^{\updownarrow}_{n, c_i^{\prec}} > 0$. The form of the eventually resulting constraint is not unique, but depends on the order in which the instructions $c^{\prec}/c^{\succ}$ were selected—therefore it is doubtful whether a closed-form formula of these tightened block length constraints exists. Their impact, however, is beyond doubt: In combination with the previously presented combined precedence constraints, they halve the solution times of the two most difficult problems encountered during the experiments (see Sec. 7.3.1).

The ILP is generated with help of ILOG Concert 2.0, a library of classes and functions that supports this process via C++ concepts [ILO03a]. The Concert framework passes the ILP directly to the solver without the indirection of a text file. The solver is then started together with several parameters that improve the solving performance significantly (more than 400% on the two most difficult input programs, compared to the default settings). The chosen parameter combination is the result of a long process of analyzing and experimenting. One highly important ingredient is the specification of suitable *branching priorities*, i.e., priorities that guide CPLEX to find the next fractional-valued variable to branch on. In line with the recommendations of Sec. 4.2.1, it turned out to be advantageous to give the B, S, and a variables a higher priority than the x variables.

It is also beneficial to emphasize feasibility over optimality via the `MIPEmphasisFeasibility` setting [ILO03b]. The solver then invests less computational effort in analyses that aid in moving the best bound and keeping the branch-and-bound tree small (see Sec. 4.2.1), but tries to find as many integer feasible solutions as early as possible. This involves also less backtracking. Our ILP model seems to be tight enough to benefit from this strategy.

The same settings and parameters are used for all input programs, there is no input-specific tuning. Also there is no optimality tolerance interval granted to the ILP solver—only a 100% optimal result is accepted. After the ILP solver has delivered such a solution, it is started again with a different objective function that minimizes the number of instructions in the schedule, as described in Sec. 6.5. The result of this second phase is then the final optimal schedule.

7.1.4 Postprocessing

Features like predication and speculation require postprocessing in the schedule, as described in Chapters 6.2 and 6.3. In addition, the solution values of the sc^A_B variables are interpreted and the necessary branches are created and inserted into the schedule. Then the basic blocks are ordered according to the ft^A_B variable values (which define the fall-through edges).

After that, all employed virtual registers are allocated to architected registers, based on a liveness analysis. If during this more registers are needed than are available in the procedure's register stack frame, the latter is extended automatically. This involves a shifting of the output area, with renaming of those registers that pass parameters to subroutines.

Each basic block is then passed to the bundler, which generates the final assembly output. Afterwards, possibly predicate register initializations have to be inserted manually into empty slots of the bundled code (the tool outputs if and where they are needed). In a few cases, empty bundles (containing only nops) are inserted manually to align hot branch targets on 32-byte boundaries. This "padding" facilitates fetching (see Sec. 2.2.3.1) and is also done by the Intel compiler. Finally and also manually, recovery code is added.

7.2 Experimental Setup

Individual routines from the SPECint 2000 benchmark [SPE00] were chosen as input of the optimizer. The method is too new and too expensive to optimize the whole SPEC suite with it; instead we had to select several routines according to the following criteria:

- The assembled routine should not have more than a few hundred instructions to limit the complexity.
- It should be *hot* enough so that the impact of the optimization can be measured. We have chosen only routines whose weight (i.e., the time spent in these routines) is at least 5%.
- It should not contain hot software-pipelined loops because software pipelining is currently not supported by the model. As mentioned above, the tool can handle these loops, but it cannot optimize them.
- It should not contain too many long-latency instructions because the model is imprecise here. This concerns mostly floating-point operations.

The selected routines were compiled to assembly with Intel's Linux compiler for the Itanium 2. We used full optimization (`-O3`) and profiling information (`-prof_use`).[4]

Table 7.1 lists the routines together with the benchmark programs they were taken from. We measure the speedup from optimizing these routines as follows: We assemble for each of them two versions of the program, one with the original version of the routine and one were it is replaced by the optimized variant. In other words, both binaries are equal except that the optimized routine is substituted into one of them. Then we run the two on a 1.4 GHz Itanium 2 and measure the execution times using the `time` utility [GNU]. The speedup of the routine is derived from dividing the speedup of the program by the weight of the routine.

[4]Following recommendations from Intel's compiler team, the first two assembly routines in Tab. 7.1 were generated with the compiler version 8 instead of 7, and with the additional flags "`-ipo`" (interprocedural optimizations across files), "`-static`" (static linking of all libraries) and "`-auto_ilp32`" (assume 32-bit address space, use 32-bit pointers).

Routine	Program	Input Set	Weight	#BB	#Loops	Ins. in
longest_match	164.gzip	program	70%	25	2	178
send_bits	164.gzip	graphic	11%	11	0	91
deflate	164.gzip	random	15%	37	3	226
firstone	186.crafty	ref	6%	8	0	37
get_heap_head	175.vpr	route/ref	30%	9	2	71
add_to_heap	175.vpr	route/ref	13%	12	1	108
qSort3	256.bzip2	graphic	12%	22	4	241
xfree	197.parser	ref	5%	9	1	46
prune_match	197.parser	ref	7%	10	1	69

Table 7.1: Input routines.

The higher this weight, the higher also the speedup of the entire program and the more precisely we can measure it. Thus, if there are multiple input sets, we perform the measurements with those input sets where the weight of the routine is maximal. The next two columns in the table list these input sets and the corresponding weights. The latter were measured with `gprof` [GNU], running an instrumented version of the benchmark program that is obtained from compiling with the "`-qp`" flag.

The remainder of the table shows the numbers of basic blocks, loops, and instructions of the input routines. All functions are optimized as a whole, except for *prune_match*, where we have omitted a large, cold part of the routine (all static numbers presented for it in the following refer to the optimized part). The "headroom" in the number of reserved cycles is for four blocks (two in *deflate* and two in *qSort3*) manually increased by one cycle (see Sec. 7.1.1). A tentative implementation of non-exclusive use forking is switched on for the routine *get_heap_head*, where it accounts for more than half of the achieved schedule length reduction. Several of the procedures contain also individual long-latency instructions (floating-point and multimedia operations), but there is only one (*send_bits*) where they lead to inter-block stalls, which are unaccounted for in the model. These stalls are manually calculated into the schedule length reduction given for this routine in Table 7.5.

The ILPs are solved with ILOG CPLEX 9.0 [ILO03b] on a Sun Fire 15000 compute server. The ILP solver task is bound to one of its 1.2 GHz UltraSPARC III+ processors via the `pbind` command for maximal performance.

7.3 Experimental Results

7.3.1 Generated ILPs

Table 7.2 shows the ILP sizes after preprocessing, i.e., after CPLEX has removed redundant rows and columns of the constraint matrix. For some input programs, the percentage of removed constraints and variables is quite large with more than one fourth and 10%, respectively. This is partly because the ILP generation relies on this redundancy detection—it is itself not specifically

Routine	#Constraints	#Variables	Relaxation Gap	1st Node Gap	2nd Node Gap	#Nodes	Sol. Time / Seconds
longest_m.	5974	3314	10.8%	10.8%	0.8%	376	111
send_bits	2497	1483	4.7%	3.4%	n/a	0	2
deflate	4450	2725	5.8%	5.8%	n/a	0	2
firstone	411	277	0.0%	0.0%	n/a	0	1
get_heap_h.	4079	1655	7.4%	6.8%	0.0%	87	15
add_to_h.	3257	1703	1.6%	0.0%	0.0%	4	3
qSort3	9836	5059	8.2%	8.1%	4.5%	709	151
xfree	657	387	15.1%	15.1%	n/a	0	1
prune_m.	1322	805	5.6%	5.5%	1.0%	13	1
		Average	**6.6%**	**6.2%**	**1.3%**		

Table 7.2: Characteristics of the generated ILPs.

optimized for the detection and removal of redundant instances of constraints.

The next three columns give in three different stages the *integrality gaps* as a measure of the tightness of the polytope. This gap is defined as the distance between the objective function values of optimal solutions of the ILP itself and the corresponding LP-relaxation (as a percentage of the first value). The gaps in the third and fourth columns of Table 7.2 ("Relaxation Gap" and "1st Node Gap") have been measured before and after preprocessing, respectively. Apparently, CPLEX can tighten the polytope slightly by strengthening bounds, reducing coefficients, etc. [ILO03b]. After that, CPLEX adds cuts at the root node of the branch-and-bound tree. They tighten the polytope considerably, as the diminished gap at the succeeding second node shows. In all except for two routines, the integrality gap is entirely closed before branch-and-bound is reached. In four cases, also optimal solutions have been found that are integral so that no branching is necessary.

The last two columns list the total numbers of branch-and-bound nodes and the solution times. As expected, the gaps are a good predictor for the solvability (except for the small routine *xfree*). The ILPs of the second phase are solved in less than one second for all input programs.

Figures 7.3 and 7.4 show breakdowns of the numbers of generated constraints and variables, respectively, averaged over all input routines. As expected, the constraint numbers are dominated by the general and local precedence constraints (Equ. (5.3.12) and (6.3.6), respectively) and the structurally similar block length constraints. The bundling constraints, included in "Others", are only generated for the first pattern in Fig. 6.15; they never contribute to more than a half percent of the total number of constraints.

Fig. 7.4 shows that the x variables account for about 80% of all variables. Compared to them the variables for the modeling of branches, for instance, are negligible. These statistics confirm the estimated high impact of the factor $\mathbb{G}$ on the ILP sizes.

7.3.2 Optimal Schedules

Before we present the schedule length reductions and speedups achieved by the optimizer, we take a closer look at the structure of the obtained schedules. Table 7.3 shows several speculation-

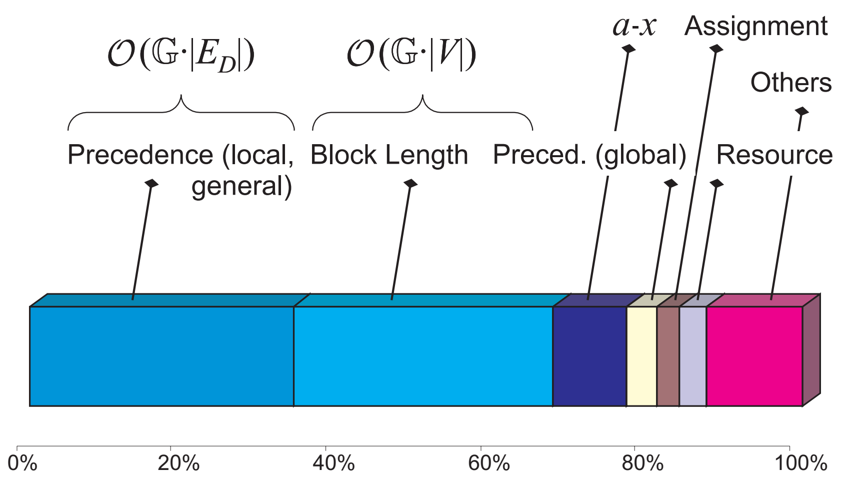


Figure 7.3: Empirical constraint distribution.

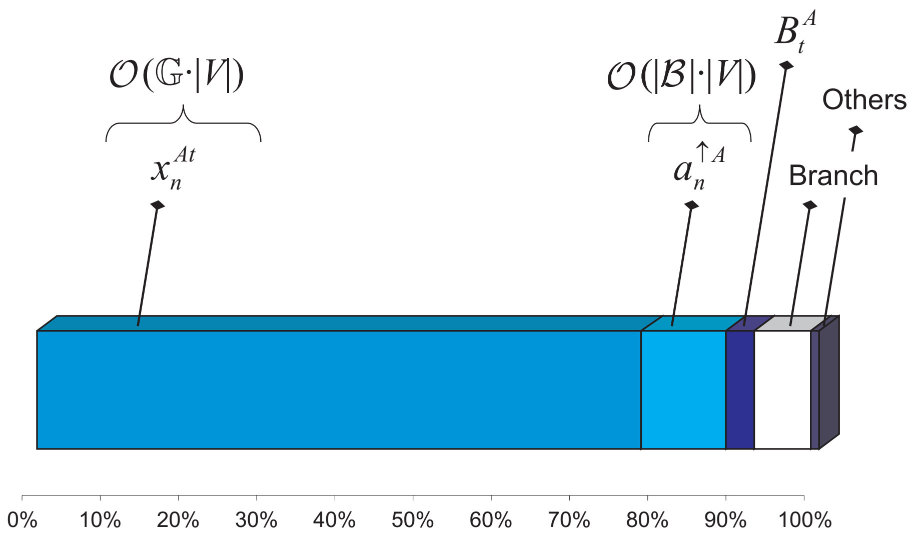


Figure 7.4: Empirical variable distribution.

related characteristics: The first two columns list the number of control or data speculative loads employed in the input and in the output schedule, respectively. In most cases, the latter uses significantly more of them. However, in only two routines more data speculation is used than before (*get_heap_head* and *add_to_heap*).

The next two columns give how many speculation possibilities according to Sec. 2.1.5 are included in the ILP (“Poss.”) and the number that is actually utilized in the output schedule (“out”). This measure comprises not only speculative loads, but also speculated concurrent definitions. The table shows that the number of mutually exclusive pairs of sets of instructions is often large (more than 40 in two cases), and also that a large proportion of them is eventually utilized (almost half on the average).

Routine	# Control/ Data Spec. in	# Control/ Data Spec. out	# Spec. Scheme Poss.	# Spec. Scheme out	Spec. Code Motion Poss.	Spec. Code Motion out
longest_m.	15	15	43	26	38%	24%
send_bits	0	1	7	1	27%	9%
deflate	4	2	28	2	21%	5%
firstone	0	5	7	5	33%	21%
get_heap_h.	3	7	23	10	33%	15%
add_to_h.	2	5	16	5	35%	16%
qSort3	7	14	46	19	38%	17%
xfree	2	3	7	4	27%	23%
prune_m.	4	7	19	12	37%	21%
Average:					**32%**	**17%**

Table 7.3: Results of the optimization: Used speculation.

The last two columns quantify the usage of speculative code motion: the first shows how many instructions *can be scheduled* speculatively according to the candidate block ranges[5], the second the percentage of instructions of which a copy *is scheduled* speculatively in the output schedule—the average value of 17% shows that the ILP solver applies speculative code motion moderately.

Table 7.4 displays in the same form how often different sorts of code motion were used: for each sort, one column always shows the percentage of instructions to which it *can be* applied in the ILP model (“Poss.”) and the subsequent column the percentage to which it *has been* applied in the output schedule (“out”) [6]. The first two columns reveal that the upward direction of code motion is used much more frequently than the downward direction. This is not surprising from the first sight: upward code motion means executing instructions earlier, allowing the schedule to end earlier.

However, from a more abstract viewpoint, upward and downward code motion are equivalent: Fundamentally, global code motion achieves schedule length reductions by overlapping

[5]This statistic does not take predicated code motion from Sec. 6.1 into account which allows an instruction to be scheduled speculatively, but never be executed speculatively.

[6]All percentages are relative to the numbers of instructions modeled in the ILP, not to the number of scheduled copies (which can be higher due to compensation copies). For example, “Upward out” gives the percentage of instructions for which at least one copy is scheduled in the output schedule as a result of upward code motion.

Routine	Upward Poss.	Upward out	Down. Poss.	Down. out	Cyclic Poss.	Cyclic out	PR Poss.	PR out	Pred. Poss.	Pred. out
longest_m.	41%	20%	9%	2%	19%	8%	13%	7%	8%	1%
send_bits	46%	12%	47%	16%	0%	0%	11%	7%	40%	8%
deflate	46%	11%	37%	5%	4%	1%	12%	4%	58%	8%
firstone	67%	21%	18%	0%	0%	0%	0%	0%	42%	0%
get_heap_h.	19%	8%	16%	6%	23%	11%	1%	1%	13%	1%
add_to_h.	48%	15%	9%	1%	13%	9%	8%	1%	22%	5%
qSort3	57%	14%	11%	2%	21%	4%	20%	12%	32%	1%
xfree	43%	16%	27%	18%	14%	7%	0%	0%	27%	18%
prune_m.	31%	16%	1%	1%	12%	8%	0%	0%	7%	0%
Average:	**44%**	**15%**	**19%**	**6%**	**12%**	**5%**	**7%**	**3%**	**28%**	**5%**

Table 7.4: Results of the optimization: Used code motion.

(parallelizing) code from different blocks. But if code from a block A can be overlapped with code from a successor block B, then it makes no difference if instructions from B are moved upwards to A, shrinking the schedule length of the former block, or if instructions from A are moved downwards to B, shrinking block A.

A fundamental reason for the subordinate role of downward code motion is that it is restricted by compares: they cannot be moved past branches they control (see Sec. 6.1.2). Since they are also often the last instruction in a long data dependence chain, they inhibit downward code motion of other instructions as well. Thus the upward direction is more widely applicable (44% vs. 19% in Tab. 7.4) and therefore used more frequently; it is often the only way to overlap code from different blocks.

The table also shows that cyclic and partial-ready (PR) code motion are limited to relatively small portions of instructions; but as it will be seen later, both are effective if applied selectively to critical-path instructions. The last two columns display results for predicated code motion from Sec. 6.1. This sort of code motion lags remarkably behind its potential (5% vs. 28%). An analysis of the schedules clearly shows that this is due to the data dependences on the controlling compares, which must be respected by instructions that use this kind of code motion. This restricts especially the upward direction since compares are, as mentioned earlier, often on the critical path and thus frequently scheduled in the last cycle of a basic block.

As a concrete example, Fig. A.3 and Fig. A.4 in Appendix A illustrate the schedules of *qSort3* before and after the optimization, respectively. In these figures the basic blocks are vertically divided into cycles and horizontally into execution slots. The filled boxes represent scheduled copies of instructions and their different colors in Fig. A.4 different kinds of applied code motion. The pictures give an impression of how code motion contributes to the schedule length reduction (which is pointed out by the double-headed arrows).

As a result of global code motion, every fourth instruction is moved out of its source block on the average over all routines. Table 7.5 lists the number of collapsed blocks resulting from this. This table also finally lists the achieved reductions of the global schedule length in the second column. Sometimes the reductions are considerable like for *longest_match* and *get_heap_head*; in other cases the critical path is a limiting factor as in *deflate* and *add_to_heap*, but the per-

Routine	Collaps. Blocks	Static Reduction	Weigh. IPC in	Weigh. IPC out	Delta Instruct.	Delta Bundles	Speedup Program	Speedup Routine
longest_m.	1	46%	2.4	5.6	24%	7%	18.50%	26%
send_bits	0	31%	2.4	4.9	10%	10%	3.00%	27%
deflate	4	19%	2.6	3.6	3%	-3%	1.72%	11%
firstone	0	37%	2.6	4.7	14%	0%	0.88%	15%
get_heap_h.	0	43%	2.3	4.9	31%	9%	4.25%	14%
add_to_h.	1	17%	3.0	4.0	11%	4%	1.17%	9%
qSort3	1	26%	2.9	4.6	16%	4%	1.93%	16%
xfree	3	22%	2.3	3.6	9%	-5%	0.76%	15%
prune_m.	2	41%	2.5	5.3	22%	-3%	0.73%	10%
	9%	**31%**	**2.6**	**4.6**	**15%**	**3%**		**16%**

Table 7.5: Further characteristics and performance of the optimal schedules.

centages are still respectable here with 19% and 17%, respectively. Later we will examine more closely how these improvements could have been achieved.

The schedule length reductions are accompanied by drastically increased static instructions-per-clock rates, as the next two columns point out: the average IPC (without nops and including branches), weighted by the execution frequencies of the blocks, goes up from 2.6 to 4.6 (the unweighted IPC from 2.4 to 3.9). This shows that the ILP scheduler is successful in extracting more parallelism and approaches the maximum IPC of six for the Itanium 2.

A further factor that contributes to the jump in the IPC is the increased instruction count by 15% on average ("Delta Ins.", without nops and recovery code). One reason for this is that the ILP scheduler can fill free issue ports uninhibitedly with speculation checks and compensation copies because it can take the execution unit occupation optimally into account in all scheduling decisions. So it can fully exploit the maximally possible IPC without running in danger of a resource oversubscription. It should also be considered that we perform global code motion on top of the global scheduling already performed by Intel's compiler. This means, for example, that compensation copies created by Intel's compiler are not fused back into single instructions again (undone), but even more of them are added. If both schedulers had the same starting point, the increase would likely be lower.

The rising number of instructions is potentially harmful to the instruction cache efficiency. But strikingly, the relevant number of bundles grows only by 3% ("Delta Bundles"). This comes from the fact that the new schedules use *fewer*, but *larger* instruction groups which fit much better into the bundling scheme of this architecture. Small groups are more often forced to be filled up with nops. In other words, most new instructions move into execution slots that were previously occupied by nops. With the small code size increase, the negative impact on the instruction cache should be negligible. The impact of growing register stack frames due to the subsequent register allocation can also be considered as minor: on the average, three more stacked registers are employed—in all optimized procedures, a reserve of more than 50 unused architected registers remains.

The second last column of Tab. 7.5 shows the measured speedups of the benchmark programs. These numbers are sometimes less than 1% because only a single small routine has been changed, therefore several runs were performed to determine them precisely. Using the weights, we derive the speedups of the individual routines from them as described in Sec. 7.2. It is visible that the runtime impact of the static improvements varies with the different stall characteristics of the routines: the performance improvements are between one third and two thirds of the schedule length reduction, with the average at half. The latter is plausible because we optimize the unstalled execution time, which is about half of the total execution time for SPECint 2000 [MK02].

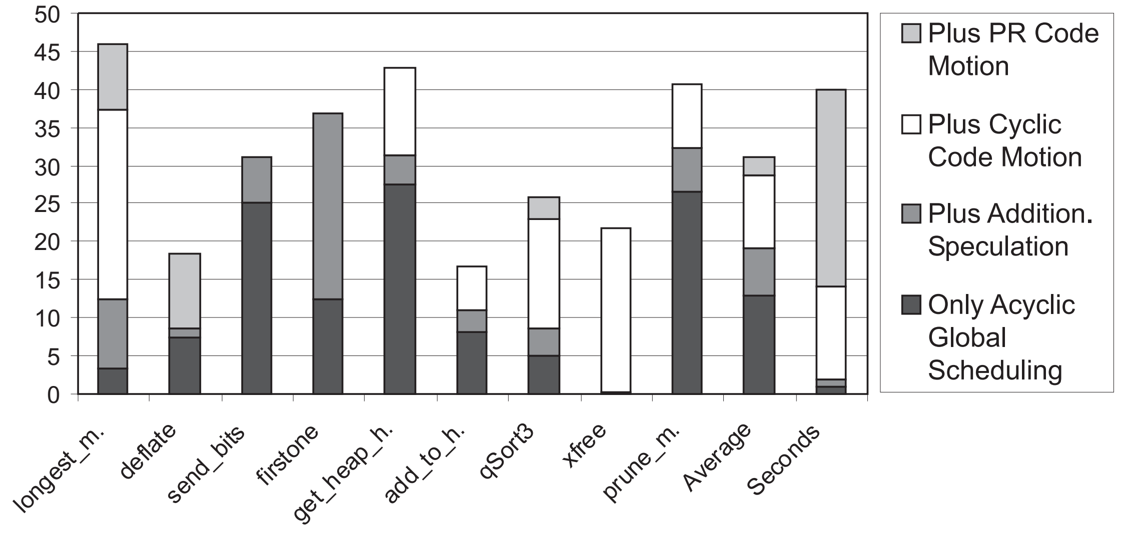


Figure 7.5: Schedule length reductions (in percent) as different extensions are switched on incrementally.

Finally, Figure 7.5 depicts how the schedules shrink as the extensions additional speculation, cyclic code motion, and PR code motion are switched on incrementally (in this order). In the setting “Only Acyclic Global Scheduling”, control speculation possibilities are included in the ILP only for those loads that are already control speculative in the input schedule. “Additional speculation” then means the full inclusion of speculation possibilities according to Sec. 6.2.

The diagram shows that all these features improve only a subset of the routines, but on the average, each is essential. The last bar shows the accompanying increase in the average solution time (the y-axis displays both percents and seconds). While scheduling plus speculation generally can be solved within a few seconds, the two other extensions cause search space expansions, especially for the two most difficult to solve routines.

Chapter 8

Related Work

This chapter is organized as follows: The first section completes the survey of global scheduling heuristics from Sec. 3.3.1. Some of the further here presented techniques are more exotic. The following main section covers exact and phase-coupled methods in code generation. These two classes are treated as a whole since many exact (optimal) instruction schedulers also incorporate decisions from other code generation phases and are thus phase-coupled. Also near-optimal or heuristic approaches to phase coupling are presented. Two subsections distinguish between heuristic and search-based methods. The latter are further subdivided into methods based on ILP, constraint logic programming, evolutionary algorithms, enumeration, and other approaches. Finally, to broaden the perspective, Sec. 8.3 outlines recent works on the use of ILP in various other compiler-related applications.

Within this chapter we focus on publications that are prominent in their field or particularly related and relevant to our work. More comprehensive surveys with further references, including historical ones, are to be found in [SS02, MG95, Leu00, Bas95].

8.1 Global Instruction Scheduling Heuristics

Many global scheduling heuristics have similarities with trace scheduling, but differ with regard to the chosen scheduling regions: In [CS98], for example, two global scheduling techniques are presented whose scheduling regions are *extended basic blocks* (traces of blocks that have not more than one predecessor in the BBG) and *dominator paths* (BBG paths between two basic blocks that are adjacent in the dominator tree [SS02]), respectively. Within these linear regions, code motion is only permitted that does not require compensation copies to limit possible code growth (the structure of the scheduling regions facilitates this for upward code motion). When compiling for an eight-issue DSP, a dynamic instruction count reduction of 5% and 6%, respectively, compared to local scheduling is reported.

In *tree traversal scheduling*, BBG subgraphs with a tree structure (*treegions*) form the scheduling scope [ZJC01]. The selection of these treegions is accompanied by tail duplication to increase their size. The basic blocks in each treegion are scheduled successively in the order of a depth-first traversal of the tree. Each block is scheduled via list scheduling, considering as candidates all instructions from this block and its successors (this incorporates speculative upward code motion). The priority function of list scheduling ensures that block-ending branches are scheduled as early as possible. Their scheduling aborts the processing of a block. All instructions that are then still unscheduled and originate from this block are moved downwards to the successor blocks (which requires duplication if there are multiple successors). They are considered again when the scheduling continues with these successor blocks. Experimental results for tree traversal scheduling show a 4% speedup over an implementation of the same algorithm that works on linear scheduling regions instead of trees and a 35% speedup over local scheduling.

Gupta proposes a completely different approach to global instruction scheduling, namely an incremental approach [Gup98]: First, an initial schedule is generated by local basic block scheduling. Then this schedule is improved by progressive applications of global code motion transformations till no further opportunity for improvements is found. Even cascaded code motions—transformations that move whole sequences of dependent instructions at the same time—are described. An implementation and experimental results are not reported. An advantage of this approach is that the code motion transformations can be easily combined with other optimizations; the paper demonstrates this for partial redundancy/dead code elimination. Furthermore, it achieves a local optimum since the improving transformations are applied until no further application is possible. Its drawback is, however, that this optimum is not global, but depends on the heuristically chosen order and shapes of the transformations. In other words, the approach tackles global code motion decisions one at a time and does not consider interactions between them, let alone interactions with other scheduling decisions.

A further class of global scheduling heuristics inverts Gupta's approach: instead of applying global code motion *after* basic block scheduling, these methods perform it first, prior to the scheduling proper. In doing so, they separate code motion decisions from fine-grain scheduling decisions. This allows to deal with the former decisions on a more abstract level and simplifies the scheduling itself, which needs only to be local. Chang et al.'s *bidirectional scheduling* [CCL$^+$96] applies first downward and then upward code motion. Subsequently, each basic block is scheduled separately using a local scheduling method. The instructions for downward and upward motion are selected on the basis of *heuristic indicators* obtained from a tentative scheduling of individual paths and blocks, respectively. For instance, in order to determine the instructions for upward code motion, each block is first scheduled locally using a special variant of list scheduling that places instructions as late as possible (without increasing the schedule length). As a result, most instructions are packed near the bottom of the schedule and typically only a few in the first cycles. These instructions exposed in the first cycles are presumably critical with respect to data dependences or resource requirements and therefore selected for upward code motion. Experiments show for individual routines speedups of more than 10% over hyperblock scheduling.

A further work by Lo et al. is unique in that it models the selection of instructions for code motion as a *min-cut problem* on a graph [LCDT96]. The approach works on pairs of adjacent basic blocks: Given a block A and a successor block B, let the "maximal moveable set" M_S contain all candidate instruction in B for which upward code motion into A is *possible* (i.e., permitted). The goal is to select a subset of M_S for which it is also *profitable*. The method decides on the selection not on a per-instruction basis, but by considering all moveable instructions at once. This is done by modeling the selection as a min-cut problem on a directed graph with node capacities[1]. This graph is basically the data dependence graph induced by M_S (with the instructions as nodes). The capacity of each node is set inversely proportional to the *estimated benefit* of moving this instruction (and all DDG predecessors) from B upwards to A, that is, smaller capacities correspond to larger benefits. Thus a node cut with *minimal capacity* corresponds to a subset of M_S with *maximal benefit* from code motion. In the graph, it constitutes the boundary between the instructions to be moved and not to be moved. The benefits of code motion are estimated using a formula that takes critical-path lengths, long latencies, as well as global register pressure into account.

The integrated min-cut approach resolves all interactions between code motion decisions with comparatively little computational effort (cubic worst-case complexity), but only isolated from fine-grain instruction scheduling decisions, whose impact is only approximated. In other words, the resulting solution is globally optimal in the space of all code motions from B to A, yet only with respect to a cost model that approximates the precise effects of code motion heuristically.

8.2 Exact and Phase-Coupled Methods in Code Generation

Phase coupling in compilers continues to be an active area of research. A recent work by Cooper et al. reports on a large experimental study of the space of compilation sequences [ACG$^+$04], i.e., the *order* in which optimizations are applied. The term "optimization" refers generally to performance-increasing transformations at all levels of program representations; in the broadest sense, this includes many of the code generation subtasks covered by this thesis.

The study focuses on largely machine-independent medium-level and low-level optimizations such as partial redundancy elimination and register coalescing. For five different programs, exhaustive enumerations of the 10-of-5 subspace of orders of optimizations were performed (sequences of length 10 drawn from 5 optimizations) and their performance impact was measured. The results show that the distances between "good" and "bad" orders can be considerable (30% and more), but that 80% of the local minima in the search space are within 5-10% of the optimal solution. However, there is also a group of unfavorable minima that are almost 30% worse. Several heuristic search algorithms are proposed (greedy, predictive, genetic) to find good sequences during program compilation.

Another work by Zhao, Childers, and Soffa approaches the problem from a completely different side [ZCS03]: Instead of a search-based approach, it develops analytic models of optimizations. In order to *predict* their performance impact, it employs *optimization*, *resource*, and *code models*, which abstract performance-relevant information about the optimization, the target machine, and the program code, respectively. To estimate the impact of an optimization, the code model together with optimization parameters is fed into an optimization model. The result is a new code model that represents the optimized code. The resource model is then applied to the

[1]The original work does not use node capacities, but we adopt here the concepts from Sec. 5.2.1 for an easier presentation.

original and the optimized code model in order to estimate the impact of the optimization. As an example of a resource model, the work presents a cache model that indicates how many cache misses a given reference pattern of a loop nest incurs. A corresponding code model abstracts the loop nest with its references, and three optimization models describe the effect of several loop transformations (loop interchange, tiling, and reversal) on the code model. The experimental results show that model-based prediction can help avoid the application of loop transformations in those cases where they degrade performance.

In principle, the two presented approaches could also be used to guide decisions concerning Itanium code generation (speculation, if-conversion, cyclic code motion). However, a fundamental difference to the ILP-based approach of this thesis is that they address *phase ordering* rather than *phase coupling*: They help determine beneficial transformations, or beneficial orders of transformations, but do not integrate different types of them. The following section gives an overview of heuristic and search-based approaches to phase coupling.

8.2.1 Heuristic Phase Coupling

Many heuristic approaches to phase-coupled instruction scheduling focus on the integration with register allocation. In [EM92], a global instruction scheduling technique with integrated register renaming is presented. Register allocation is performed prior to scheduling. If during scheduling a false dependence due to a destination registers obstructs the placement of an instruction, the destination register is renamed to a free temporary register (if existing). A move operation is added to transfer the value in the temporary register back to the original register. The procedure is similar to our speculation scheme for concurrent definitions in Sec. 6.2.1.

A deeper integration is achieved by *integrated prepass scheduling (IPS)*, a classic approach in which instruction scheduling precedes register allocation [GH88]. The scheduler keeps track of the number of available registers at the current cycle. An instruction scheduled into this cycle can decrease (by creating a new live register) or increase (if it terminates a live range) this number. The basic idea of IPS is that the latter group of instructions is given a higher priority if the number of available registers falls below a threshold. The subsequently following global register allocation phase needs to introduce fewer spills and fills since the number of concurrently live registers in the schedule is limited. Further references and studies regarding heuristic coupling of instruction scheduling with register allocation can be found in [SS02, Bas95, BGS98].

In [BCGS96], an approach is presented that integrates selected optimizations into the scheduler. These optimizations have in common that they are not always useful, or more precisely, that their usefulness depends on information available not until scheduling, such as the availability of registers or functional units at a program point. The optimizations include, for example, redundancy elimination (which can significantly increase register pressure) or DDG restructuring transformations that exploit algebraic properties of operators to reduce the critical path length (they may lead to increased register and execution unit requirements). These transformations are applied during scheduling only if there is no shortage of available registers and execution units.

The integration proposed in [BCGS96] is closely related to *mutation scheduling*, a technique developed by Novack and Nicolau [NN94]. Mutation scheduling combines instruction scheduling (based on percolation scheduling, see Sec. 3.3.1), code selection, and register allocation into a single phase. It is a "value-oriented" approach in that it "allows the computation of any given value to change dynamically during scheduling to conform to varying resource constraints and availability" [NN94]. More concretely, for each value in the program there exists a *mutation set*, which refers to multiple alternative expressions (typically instructions) that compute this value. For example, an address computation for an array reference often requires a multiplication with a constant, which can be done either by using a multiplication instruction or a series of adds and shifts—which of these computation alternatives is preferable during scheduling depends on factors like the availability of a multiplication unit.

The expressions are also called *mutations* and may themselves refer to other mutation sets, so that a tree structure emerges. They change dynamically during scheduling as new values become available: For instance, if an instruction is scheduled that writes a value into a destination register, then this register is added to the value's mutation set. Similarly, if a value is spilled, then the corresponding fill instruction (load) is added to the set. In this way, the decision whether to reload or to recompute (*rematerialize*) a value is also incorporated. The choice of one mutation out of the mutation set during scheduling is guided by a heuristic selection function that weighs the relative merits of each, subject to the available resources at the current cycle (determined by the target architecture and the partial schedule).

In relation to our work, the mutually exclusive instruction sets from Sec. 6.2 can be considered as mutations. But unlike the *complete* phase coupling achieved by our ILP-based approach, the *heuristic* phase coupling through the greedy selection function of mutation scheduling does not take interdependences between mutations of different values into account (there is no backtracking). These interdependences are strong since each scheduling of a mutation that computes a value influences all other mutation sets that refer to this value (see above). It also affects the available resources and thereby the selection function.

8.2.2 Search-Based Methods

8.2.2.1 Based on Integer Linear Programming

The earliest ILP models for phase-coupled code generation emerged in *high-level synthesis*, the automated generation of hardware circuits from a high-level input description. The objective of high-level synthesis is to generate a VLSI hardware architecture that implements a specified behavior while satisfying a set of constraints and minimizing a cost function. More concretely, possible goals are to minimize the execution time under given hardware constraints, or to minimize a cost function of the functional units (e.g., chip area) subject to an upper bound on the execution time. The input program can be given by a CFG and a DDG; the output is a hardware description together with a corresponding code sequence. The main tasks of the synthesizer include instruction scheduling, resource binding, and register allocation.

The synthesizer ALPS is one of the first to employ ILP for scheduling and resource binding [HLH91]. Its ILP model uses time-indexed binary variables to model scheduling decisions and is extended to minimize the number of buses or the live ranges of registers. Experimental results, however, are only shown for one small example (a basic block with 34 instructions). It is noteworthy that this early approach features already an ASAP/ALAP analysis to minimize the number of decision variables. In contrast to ALPS, the synthesizer JOSHUA models the cycle at which an instruction is scheduled directly via an integer variable; an additional set of binary variables defines the functional unit where it is executed [WGB94, WGHB95]. The ILP formulation allows to synthesize multiple basic blocks at the same time (without global code motion, however) and to take code selection alternatives as well as register allocation into account. Experimental results are only given for one very small routine with 24 operations [WGB94]—the choice of integer variables for the main scheduling decisions makes the ILP formulation complicated and arguably inefficient. Most later approaches use binary variables instead.

The apparently very limited practical applicability of these early attempts is partly the result of modest ILP solver technology at this time, but it is also due to the fact that the employed ILP models were *ad-hoc* formulations whose efficiency was not analyzed and optimized at all. In particular, they often used the following ad-hoc formulation of the (local) precedence constraints:

$$\sum_{t_m \in G(A)} t_n x_m^{At_m} + w_{mn} \leq \sum_{t_n \in G(A)} t_n x_n^{At_n} \qquad \forall (m,n) \in E_D, \forall A \in \Theta^x(m) \cap \Theta^x(n) \qquad (8.2.1)$$

These constraints do, in contrast to Equ. (5.1.18), not describe an integral polytope, which may impair—especially for large inputs—the solvability. Nevertheless, they also have advantages over (5.1.18), namely that a far lower number of them is generated and that they are more straightforward.

Gebotys and Elmasry [GE92, GE93] are among the first to analyze and advance the polyhedral structure of their ILP model: They reformulate high-level synthesis as a node packing problem and show that some of their constraints represent (under circumstances) integral facets. For example, this is done for the precedence constraints which have the same form as (5.1.18). As a result of the well-structured formulation, their OASIC synthesizer achieves solution times of less than 90 seconds for basic blocks with more than 100 instructions. These numbers do include resource binding, but not register allocation for which an additional modeling is given. A further synthesizer, OSCAR [LMD94], implements an ILP model that is based on OASIC and incorporates further features such as varying latencies of instructions on different functional units and support for complex components with internal chaining (like fused multiply-and-add units).

Chaudhuri et al. conduct for the first time an extensive mathematical analysis of the polytope [CWM93, CWM94] described by their ILP model of local instruction scheduling (which is very similar to the OASIC model). They prove that the two subpolytopes of the assignment constraints combined with either the precedence constraints or the resource constraints are polynomial sized and integral (see Fig. 5.18). For the latter polytope, this is shown by means of total unimodularity; for the former, the proof is done by reducing the corresponding subproblem to the node packing problem on a perfect graph—this thesis builds on the same approach and extends it towards global scheduling (see Sec. 5.1.1). Moreover, in [CWM94] a description can be found of how the resource constraints can be made stronger in the total polytope by selectively increasing the coefficients of their x variables from one to larger integers. This tightening could also be applied to the ILP model of this thesis and further improve its solution efficiency.

Apart from the use of ILP-based methods in high-level synthesis, there have been several approaches to integrate them into the code generator of a compiler. In [CCK97] an ILP model for simultaneous local instruction scheduling and register allocation on a superscalar processor is presented; the ad-hoc nature of the model (including precedence constraints of the form (8.2.1)), however, leads to disproportionate solution times such as 20 minutes for a simple 10-instruction example.

To demonstrate the practicability of optimal instruction scheduling in a product compiler, Wilken et al. incorporate an ILP-based scheduler into the GNU `gcc` compiler [HLW00, GNU]. Their time-indexed ILP model comprises local scheduling without phase coupling and uses the ad-hoc precedence constraints (8.2.1); to tighten the polytope resulting from these constraints, further "dependence cuts" are added. The ILP-based scheduler performs extensive precomputations in order to minimize the size and the number of the ILPs to be solved: For instance, it tries to partition the scheduling region into smaller subregions that can be dealt with separately without losing the global optimality of the combined solution. A further transformation called region linearization is applicable only because the work targets a single-issue processor.

Basic blocks are not passed to the ILP solver if a lower bounds show that they have already been optimally scheduled by the heuristic scheduler (which implements critical-path list scheduling). As a result, in the experiments ILPs are generated only for 22 out of more than 7,000 basic blocks; they are solved in a total of 45 seconds. In the entire program, the ILP scheduler saves a total of 66 static cycles. The runtime performance impact is not reported. The work shows that optimal basic block scheduling is feasible, but also that it delivers no extensive improvements over list scheduling—at least for the targeted single-issue processor with a three-cycle maximum latency[2]. Earlier studies confirm that the results of local list scheduling are generally close to the optimum [CSS98].

This is different for digital signal processors (DSPs), which often have irregular and highly dedicated microarchitectures. There the quality of the code generated by heuristic code generators is often insufficient, which is problematic since the domains of embedded processors are characterized by severe cost restrictions (and often also power constraints) [Käs00a]. Thus the use of ILP-based, phase-coupled code generators is promising here. The retargetable RECORD compiler for DSPs employs an ILP model for local code compaction[3] that takes encoding conflicts into account and allows to select among alternative encoding versions [LM97]. The authors report solution times of a few seconds for basic blocks with 23-45 operations.

All the ILP-based code generation approaches mentioned so far have in common that they do not incorporate global code motion. PROPAN, a retargetable framework for postpass optimizations, lifts this restriction [Käs00b, Käs00a]: Its ILP-based instruction scheduler allows code motion between *control equivalent* basic blocks inside *superblocks*, scheduling regions that are similar to the traces of trace scheduling (see Sec. 3.3.1). Hence the scheduling regions may not contain disjoint control flow paths and no code motion is possible that is speculative or requires

[2]Remarkably, the same problem with a *two*-cycle maximum latency is already polynomial-time solvable [HLW00].

[3]I.e., minimizing the code size. On DSPs, this means minimizing the number of instruction words, which basically corresponds to the goal of instruction scheduling.

compensation copies. However, the superblocks may exceed loop boundaries so that code motion across these boundaries is possible.

PROPAN is based on two alternative, well-structured ILP models for phase-coupled instruction scheduling, resource binding, and register allocation: one is based on the above-mentioned, time-indexed OASIC formulation and the other on the order-indexed SILP formulation (more on the latter below). To model global code motion, the scheduling region is regarded as a single, contiguous, and linear sequence of cycles that comprises all the basic blocks in the region. This is possible since the order of the blocks within the superblock (trace) is fixed and sequential. The mapping of cycles in the sequence to basic blocks is determined dynamically in the ILP via two integer variables t_i^A and t_i^E for each block b_i: $t_i^A \leq t \leq t_i^E$ implies that cycle t belongs to block b_i. In this way, the basic block boundaries can be moved dynamically within the modeled sequence of cycles so that the same cycle can be mapped to different basic blocks. This is more flexible than the approach followed by this thesis where each modeled cycle is defined to belong to only one fixed basic block. Here a dynamic partitioning would be advantageous as it would ease the difficult choice of the values $\mathbb{G}_A$, but unfortunately, it requires linear scheduling regions and cannot be combined with disjoint control flow in the scheduling region (this is also noted in [Käs00a]).

Experiments are reported with 19 routines from the `dspstone` benchmark comprising 9-95 operations. For experiments with the TriMedia TM1000 processor, these routines are compiled to assembly with Philips' `tmcc` compiler. The postpass optimizer PROPAN can reduce the schedule lengths of five of these assembly routines (four of them by more than 10%). When targeting the ADSP-2106x SHARC processor, the improvements are larger with 8% on the average.

When restricting the optimization to loop boundaries, the BBGs of the routines are covered by 1-5 superblocks, which are optimized successively. This occurs for most routines in less than one minute using the OASIC model (targeting the TriMedia). Register allocation is excluded since it requires an exponential number of constraints in the OASIC formulation. If the superblocks are allowed to cross loop boundaries, most of the routines are covered by a single superblock and hence optimized as a whole. The optimization results do not change, but the computation times are much higher and reach several minutes and hours; no ILP-based solution could be obtained for 4 large routines within 8 hours (the largest ILP has 68432 constraints and 11562 variables).

Alternatively, the *order-indexed* SILP formulation can be used in PROPAN, which permits a more efficient integration of register allocation [Zha96]. Here the main decision variables do not map the instructions to cycles, but describe instead an imaginary flow of the execution units through the instructions (which constitute the nodes of a resource flow graph). Concretely, there are flow variables x_{ij}^k that are equal to one if and only if an instance of the execution unit type k flows from instruction i to instruction j (then j is executed after i on k). Since integral polytopes for network flow problems are well known (see Sec. 5.2.1), a subpolytope of the resource constraints with this property is easily found. In contrast, the precedence constraints cannot be directly formulated on the flow variables since the latter define only the order of instructions that are executed on the same type of execution units. Thus there exist further integer variables that explicitly hold the cycles where instructions are scheduled, e.g., the variable t_i is equal to the cycle of instruction i. The precedence constraints can be formulated easily and very efficiently on the basis of these variables, but the also required *serial constraints*, which synchronize them with the flow variables, are a weak point in terms of structural efficiency (as typical for constraints that connect integer variables to binary variables) [Zha96].

The experiments in [Käs00a] show that SILP is well suited for irregular architectures with little parallelism, while the time-indexed OASIC model is preferable for architectures with a high-degree of instruction-level parallelism. Thus we have based our model for the Itanium Processor Family on the latter formulation.

8.2.2.2 Based on Constraint Logic Programming

There are several approaches to model phase coupling with *Constraint Logic Programming (CLP)*, a more general method than ILP: Here the values of the variables can be from arbitrary domains and the constraints can exclude arbitrary combinations of variable assignments. A solution of a *constraint satisfaction problem* is an assignment that meets all constraints. As constraints are added, CLP systems use *constraint propagation* to reduce the domains of the variables (and consequently the search space) by excluding values that will lead to infeasible solutions. Eventually, a constraint solver *labels* the variables with values of their domains until a complete, globally feasible solution is found (which is an $\mathcal{NP}$-complete problem). Similar to ILP, this solving procedure involves backtracking and can be guided by search strategies. If the structure of the constraints permits it, even an ILP solver can be used.

Such an hybrid approach is employed by Gebotys for simultaneous instruction selection, compaction, and register allocation [Geb97]: the problem is formulated as a set of logical propositions, most of which are *Horn clauses*. These propositions are translated into an ILP with binary variables. The rationale behind this translation is the following property: if all of the propositions are Horn clauses, then already the LP-relaxation delivers an optimal integral solution (if non-integral variable values are rounded to their nearest integer values). This leads to short solution times (if scheduling is excluded), however, the model is only suitable for very restricted architectures since its complexity is exponential in the number of modeled registers.

Van Beek and Wilken describe a CLP-based local instruction scheduler for single-issue processors with arbitrary latencies [vBW01]. To assist the constraint solver, they add several redundant constraints that reduce the domains of the variables via constraint propagation. These auxiliary constraints take advantage of the structure of the scheduling problem, in particular by exploiting that the instructions must be serialized due to the single-issue restriction. As a result, a standard constraint solver can optimally schedule the basic blocks almost without backtracking, 22 times faster than Wilken's ILP-based scheduler [HLW00] (see Sec. 8.2.2.1). But as with the ILP-based approach, these results rely on the single-issue limitation and it is unclear to which extent they can be transferred to contemporary superscalar processors.

A more comprehensive constraint driven code generator is presented by Bashford and Leupers [BL99b, BL99a]. It integrates code selection, local scheduling, and register allocation and targets embedded processors with highly restricted and irregular data paths like the Analog Devices ADSP-210X. On this DSP there exists no general register file, but only special-purpose registers that are connected to the inputs and outputs of specific functional units. Thus there are strong interdependences between code selection and register allocation—and also between register allocation and instruction scheduling since some operations can only be executed in parallel if they access certain registers.

To enable a tight coupling between these tasks, decisions on code selection and register allocation are *delayed* as long as possible. Instead of finalizing them heuristically, a search space of possible alternative decisions is described by a constraint system. An initial set of constraints is prescribed by the target architecture. The code generation phases add further constraints and variables, which reduce the search space. The achieved phase coupling is not *complete*: some decisions are made by heuristics, e.g., scheduling is performed by list scheduling. But this algorithm does not schedule concrete, finalized instructions, but *spaces of alternative instructions* with sets of alternative *storage resources* (registers, memories) as operands. The scheduling decisions are represented by additional constraints that may reduce these spaces. In this approach, code is not *constructed*, but "obtained by a successive *reduction of the solution space* based on the constraints imposed by the target processor" [BL99b].

Eventually, labeling by the constraint solver yields the final code by finding one concrete solution in the search space. However, it is not guaranteed that such a globally feasible solution exists since only local feasibility is checked when constraints are added. If the labeling fails, constraints can be relaxed—but this remedy was not necessary in the experiments. The implementation is based on the CLP language ECLiPSe and targets the above-mentioned ADSP-210x. It generates assembly code for four routines from the `dspstone` benchmark that is more than twice as compact as code obtained from a GNU compiler; the quality comes close to handwritten code. The overall compilation speed is 3-5 generated instructions per second.

8.2.2.3 Based on Evolutionary Algorithms

Evolutionary algorithms (EA) mimic natural evolution in order to solve optimization problems. The *population* of an EA consists of several *individuals*, each of them representing a solution of the optimization problem. Each individual has a *chromosome* that contains several *genes*. These genes can be regarded as sets of variables representing the solution. A concrete assignment of the gene's variables is referred to as an *allele*. An EA applies genetic operators like *selection*, *mutation*, and *crossover* to remove, randomly modify, and combine members of the population, respectively. The purpose of selection is to prune the least promising solutions and to ensure that only the *fittest* individuals evolve. Such continuous *evaluation* increases the chance that after a certain number of generations a near-optimal solution can be found within the population. Thus EAs are not exact, but heuristic, search-based solution methods.

In [LM04] an EA-based approach to phase-coupled code generation for DSPs is presented. It integrates code selection, local instruction scheduling, and register allocation. The input basic block is given as a data flow graph (DFG), which is a type of data dependence graph. Its nodes represent operations and its edges true dependences. Additional nodes are inserted to model possible explicit data transfers. In the used chromosomal representation, each gene corresponds to a DFG node. Its variables describe all feasible combinations of types and operand registers of an instruction that can execute the operation of the node.

The algorithm first generates an initial population. For this, multiple individuals are instantiated whose genes describe different, concrete solutions of the code generation problem. These solutions are generated by list scheduling with *probabilistic* code selection and register allocation on the fly. It is ensured that only feasible resource combinations are selected. The evaluation function favors shorter schedules and among schedules with the same length those with fewer memory accesses. The crossover operator plays a pivotal role during the evolution process: It generates new individuals by probabilistically swapping genes between two selected individuals. The crossover operator used in the work swaps only those genes that are assigned to cycles greater than a certain value (in other words, it swaps only the tails of the schedules); the authors report that this leads to faster convergence. The subsequent mutation operator checks the correctness of the results and introduces randomized variations.

In the implementation, the population size is fixed at 30 and the number of generations to be explored is set to 8 times the number of DFG nodes. Results for two DSPs are presented: On 15 `dspstone` benchmarks, the genetic code generator achieves average schedule length reductions of 51% for the M3-DSP and 38% for the ADSP-2100, compared to a standard implementation with tree-based code selection and restricted phase coupling. The compilation times range from several seconds to several minutes.

Cooper et al. uses an EA to find optimization sequences that yield a reduced code size [CSS99]. The setup is very similar to the experimental study reported on in Sec. 8.2. The experimental results show that the code compiled with a *program-specific* order of optimizations computed by the EA is often smaller (by more than 10% for almost half of the benchmark programs) than the compiler's fixed default order. However, the compilation required about one day for most programs (exploring 1000 generations). A further experiment shows that the EA finds a solution of the same quality significantly faster than mere random probing of the solution space. Further works on EA-based code generation can be found in [Bea91, Bli96, ZTB00].

8.2.2.4 Based on Enumeration

There have been various approaches to obtaining optimal or near-optimal through *implicit enumeration*. An *exhaustive (explicit)* enumeration computes recursively all possible schedules and selects one that is optimal with respect to an objective function. However, this brute force approach is impractical since the resulting enumeration tree is of immense size even for small basic blocks. Therefore, as with branch-and-bound in integer linear programming, techniques to prune the tree are essential. Any node in the tree represents a partial schedule. The enumeration can be stopped at a node if a *lower bound* shows that no completion of the corresponding partial schedule can yield a schedule that is preferable over the best solution found so far. This reasoning can be regarded as an implicit enumeration of all solutions in the subtree rooted at this node. When optimizing for schedule length, such lower bounds can be taken from optimal schedules that ignore the resource or the precedence constraints. The lengths of such schedules can be computed relatively easily; the former is equal to the length of the critical path in the DDG.

A local instruction scheduler for IA-64 based on recursive enumeration is presented in [HB01]. At each step of the recursion, all possible instruction groups are chosen from among the set of ready instruction. Each selection is bundled (using deterministic template choice) and a recursive call is performed to find an optimal schedule of the remaining instructions (if various pruning criteria do not apply). The scheduler does not target a specific microarchitecture and generates arbitrarily large instruction groups. Thus the resulting schedule is—in contrast to the approach of this thesis—not resource-constrained so that its length is predetermined by the length of the critical path in the DDG. The performance loss from not taking the dispersal into account is unclear since no dynamic results are given. Optimization goal is a static measure, namely to minimize the number of nops. The experiments show that the method produces more than a third fewer nops than a greedy scheduler and bundler, even if the search space is significantly heuristically curtailed and if a one-second timeout per basic block is imposed. These restrictions lead to a loss of optimality, but they also manage to limit the compile-time overhead to a few percent.

In [NR01] a refined branch-and-bound technique for instruction scheduling is described that also allows for resource constraints. In addition to determining a lower bound on the minimal schedule length of a subtree, also an *upper bound* is computed via list scheduling. If both bounds are equal, then the solution obtained from list scheduling is already known to be optimal so that the subtree does not have to be explored. Results are presented for several DSP floating-point routines, each consisting of one basic block with up to 100 instructions. Optimal schedules for several imaginary target processors with different numbers of ALUs and multipliers are computed with both the authors' and an ILP-based scheduler (using the OASIC formulation). While the former delivers most solutions in less than 0.1 seconds, the latter requires considerably longer. This is arguably to a large extent due to the used ILP solver `lp_solve`.

A very similar approach is taken in [LC02] to generate code for a tightly constrained and irregular network processor. Here also decisions concerning code selection and register allocation are integrated into the branch-and-bound process. As a result, the method is only suitable for very small basic blocks; the average number of operations per basic block in the experiments is 9. The enumeration is aborted if the branch-and-bound tree reaches 100,000 nodes, a threshold that was hit on approximately 10% of all basic blocks. The threshold guarantees solution times of not more than a few seconds per block, but it also leads to not always provably optimal solutions. Nevertheless, the resulting schedules are 13-15% shorter than those generated by a list scheduling implementation.

Keßler and Bednarski propose to lift the limitation of schedulers based on enumeration by combining them with dynamic programming [KB01]. The basic idea is that two partial schedules reached during enumeration are equivalent if they contain the same instructions and have the same *time-profile*. The latter describes the occupation status of the execution units at the end of the partial schedule, information that may influence future scheduling steps that extend this schedule. During branch-and-bound, two equivalent schedules can be reached over different enumeration paths—then the corresponding nodes in the branch-bound tree can be merged so that the tree becomes a graph. This enables a reuse of partial solutions as known from dynamic programming. The equivalence is in the implementation determined via hashing. Experiments with several hundred random DDGs show that inputs with up to 20 operations can often be scheduled optimally in less than one second and inputs with up to 30 operations in less than one minute. Problems with 50 or more operations may take several hours, depending on the structure of the DDG. These results include code selection and allow for a limited number of registers by discarding partial schedules whose register need exceeds this number. By comparison, a naïve enumeration algorithm becomes, according to the authors, already impracticable on instances with more than 15 operations.

A particular advantage of approaches based on enumeration is that they can handle the idiosyncrasies of irregular architectures well since they can be combined with customized routines that check the feasibility of partial schedules. Architectural restrictions may even be advantageous in that they reduce the search space [LC02]. Nevertheless, the presented works indicate that instances with a hundred and more instructions may be difficult to solve to optimality. *Backtracking schedulers*, which may revise individual scheduling decisions by *unscheduling* instructions, can be considered as a limited form of enumeration. They can be employed to deal with specific weaknesses of greedy schedulers, for instance, to achieve an effective utilization of branch delay slots [BMA02].

8.2.2.5 Other Approaches

In [MF02] resource-constrained local instruction scheduling is formulated as a *satisfiability (SAT) problem*. Given a scheduling problem instance, CNF formulas F_t are constructed that are satisfiable if and only if a schedule with not more than t cycles exists. The algorithm first sets t to the length of a schedule obtained from list scheduling and then decrements it until F_t becomes infeasible—then it is proven that the solution of F_{t+1} corresponds to a minimal-length schedule (this search direction is chosen because infeasible instances are much more difficult to solve than feasible instances). Despite the high complexity of the resulting formulas (more than 70,000 clauses for the largest DFG in the experiments with 42 operations), the solution times in most cases do not exceed a few seconds or minutes. In comparison, an ILP model that uses exactly the same binary variables is reported to yield significantly longer solution times; however, the structure of the employed ILP constraints is not described.

A further work tackles scheduling and resource binding in high-level synthesis with *symbolic scheduling*, an approach based on symbolic model checking principles [CLL$^+$02]. Based on a scheduling automaton, the set of all minimum latency schedules is symbolically computed. Each of these schedules is symbolically associated with all possible valid allocations of resources, so that the combined space can be explored for an allocation that is minimal with respect to a cost function. The experiments demonstrate that instances with 12-78 operations are tractable with memory usages below 15 MB and solution times of less than 2 minutes.

8.3 Other Recent Work on ILP-Based Compiler Optimizations

Besides scheduling, a classic area of ILP-based compiler optimizations is register allocation. Several recent works tackle this code generation subtask, which is also $\mathcal{NP}$-complete [SS02]. In [FW02], an optimal global register allocator is presented that is faster than previous work since it applies reduction techniques to identify potential register deallocation and spill decisions in advance that are unnecessary for an optimal solution. This allows to exclude these decisions from the ILP. Given a time limit of 7 seconds per function, the ILP-based allocator performs optimal register allocations for 2202 out of 2399 functions from the SPEC92 benchmark. With a 1024 second time, it succeeds with 2362 functions and inserts in these functions 3.5 times fewer loads and stores than a heuristic allocator.

Another work by Appel and George focuses on optimal generation of spill code for IA-32, which is a relevant problem since this architecture has only six allocable registers [AG01]. In comparison with a heuristic standard allocator, the ILP-based allocator requires 1522 versus 45 seconds during compilation, but the Pentium code it generates reloads less than half as many variables and runs almost 10% faster. The approach from this work is carried over to a completely different, specialized architecture in [GB03], namely to an Intel IXP network processor. The ILP model presented there deals with multiple heterogeneous register banks, register aggregates (sets of up to eight adjacent registers that are accessed as a whole by memory instructions), and variables with multiple simultaneous register assignments.

A further work addresses the efficient generation of spill code for the StrongARM processor [NP03]: there exist memory instructions that transfer two 32-bit register values in parallel via the 64-bit memory bus from or to two consecutive memory locations in SDRAM. In order to utilize these combined loads and stores for spills and fills as often as possible (saving more than 40 cycles at a time, compared to two separate instructions), the scalar variables have to be rearranged in memory in such a way that consecutive spills and fills in the code can access consecutive memory locations. An ILP-based solution to this $\mathcal{NP}$-complete placement problem is reported to yield an average performance improvement of 4.5%.

Naik and Palsberg [NP04] present an ILP-based, code-size-aware register allocator for Zilog's Z86E30 microcontroller. The register file of this processor is organized into 16 banks of 16 registers each. One of these banks can be specified dynamically as a "working bank" via a special `srp` instruction. In the encodings of certain instructions (about 30% in the used benchmark), registers can be identified either via a 4-bit number—if they are in the working bank—or via an 8-bit number else. Naturally, the former variant is preferable in terms of code size. Thus the objective of the developed ILP model is to distribute the variables among the different banks and to insert `srp` instructions in such a way that the 4-bit encodings can be used as often as possible.

In the experiments, entire programs with up to 949 instructions and up to 48 procedures were optimized. The resulting code is reported to be nearly as compact as handwritten code. However, the ILPs of the largest programs can only then be solved within a few hours if the insertion of the `srp` instructions is not allowed at *any* program position, but only at entry and exit points of procedures and interrupt handlers. Judging from the smaller programs, this restriction degrades the quality of the solution only slightly.

Chapter 9

Conclusion and Outlook

In this thesis we have tackled global instruction scheduling for the Itanium 2 processor using integer programming. Our ILP model comprises speculative, partial-ready, and cyclic code motion with automated insertion of compensation code, support for control/data speculation and predication. To the best of our knowledge, we are the first to provide an exact solution to this problem. An extensive polyhedral analysis has proven the high efficiency of the developed ILP model. This is also confirmed by the reasonable solution times in practice. In the experiments, ILP-scheduled code has exhibited substantial performance improvements over a state-of-the-art product compiler. The following listing recapitulates the primary achievements of the dissertation in more detail:

- We have presented a *formal definition* of global instruction scheduling with *speculative code motion* and *compensation copies*. The simplicity of this definition is enabled by a *path-based view* of the problem.

- Based on this formal definition, we have identified different *constraint classes* and formulated several *subproblems* of global scheduling. Then we have developed *polynomial sized* ILP formulations for PCGS-B (the subproblem without resource and block length constraints) and for the resource constraints. We have proven their correctness as well as that both describe *integral* subpolytopes:

  – For PCGS-B, this has been done by reducing this subproblem to a *node packing problem* on a perfect graph and exploiting a well-known result in order to obtain an integral polytope. The size of the obtained formulation is then reduced from *exponential* to *cubic* by removing redundant constraints and *lifting* the polytope to a higher dimension (while preserving its integrality).

  – The resource constraints are based on a *resource flow network*, a novel, expressive description of the resource binding on a processor. Initially, they describe an integral and polynomial sized subpolytope, but introduce several auxiliary variables. To remove them, the subpolytope is projected (in an integrality-preserving way) onto a lower-dimensional subspace. The resulting *network inflow resource constraints* do not require additional variables. However, their worst-case number is exponential (we prove that no polynomial sized description of the projection exists), but this asymptotic complexity does not materialize on the Itanium 2.

- As a central theoretical result of this thesis, we have proven that the identified integral and polynomial sized subpolytopes are *maximal* in the sense that they cannot be extended by other constraint classes without losing one of their two efficiency properties (integrality and polynomial size). This follows—under the assumption $\mathcal{P} \neq \mathcal{NP}$—from two $\mathcal{NP}$-completeness proofs for the extended subproblems. It substantiates that the found ILP model is close to maximal efficiency.

- Several extensions of the model have been developed that incorporate further variants of code motion and speculation:

  - *Predicated code motion* extends the candidate blocks of non-speculative instructions via predication. A further add-on models the changes in the branch structure resulting from collapsed blocks. The combination of both extensions includes if-conversion decisions into the search space.
  - We have presented a speculation scheme that combines *control speculation* with *speculation of concurrent definitions* and implemented it on the basis of mutually exclusive sets of instructions. Decisions on *data speculation* (with or without recovery code) are incorporated into the same scheme. The data speculation failure penalties can be taken into account on several levels of precision, depending on the detailedness of available aliasing probability estimates. *Non-exclusive use forking* is outlined as a novel transformation that shortens the schedule by duplicating and speculating instructions from disjoint control flow paths.
  - *Partial-ready code motion* is included, which allows to move instructions further upwards along a path by speculatively ignoring data dependences on instructions from other paths. Unlike earlier work [BMM00], we also develop a formal definition of this transformation.
  - To increase the scheduling scope, the model is extended to support *code motion into and out of loops* (that are not software-pipelined). Upward code motion out of loops includes *cyclic code motion* in the opposite direction of the backedges, a variant that can reduce the schedule length inside the loop body.

- A *bundler* has been developed that computes bundle sequences of minimal size by means of precomputed results and dynamic programming. It is assisted by *bundling constraints* in the model, which prevent structurally infeasible instruction groups.

- The *experiments* were conducted with a *postpass tool* that implements the ILP scheduler. It performs various precomputations to reduce the search space and tightens constraints dynamically during the ILP generation. Using this tool, we have optimized a selection of hot functions from the SPECint 2000 benchmark. Each is scheduled as a whole; the solution times vary from a few seconds for smaller inputs to a few minutes for the largest routine with 241 instructions, 22 basic blocks, 4 loops, and 46 speculation possibilities. The achieved global schedule length reductions relative to Intel's Itanium compiler range approximately between 20-40%. The measured speedup of the routines is 16% on average. An analysis shows that the extensions contribute heavily to these improvements, but also to increased solution times.

- Various results of this thesis, and of related research, have been presented in a number of publications [Win04, SW04, Win02, KW01].

It is important to see the experimental results in the context that the era of low-hanging fruit in processor performance is over: According to conversations with compiler developers, today already optimizations are pursued that deliver merely a 1% gain if it is *across the board*, i.e., if it shows up in most of the target benchmarks. Our tested routines may be too few and not representative enough to draw final conclusions, but the extent of the improvements in both schedule length and IPC indicates that there is still some considerable performance headroom in tasks that are very fundamental to EPIC: static scheduling and use of speculation.

Although the ILP method is—because of the relatively long solution times—not suited for today's product compilers, it is promising as a stand-alone *optimization tool* for compute-intensive application kernels like compression and encryption routines. A further interesting application is as a *research tool*: The comparison of the optimal results with those of heuristics indicates and quantifies room for improvements in the latter. It helps determine where the search for improvements can be worthwhile and—not less valuable—where not.

Moreover, the ILP approach can provide theoretically well-founded *insights into the performance potential* of different EPIC microarchitectures: In contrast to scheduling heuristics, it is simple to model architectural restrictions and asymmetries with this method and to obtain schedules that account for them optimally. It permits to *factor out* compiler influence (related to the modeled code generation subtasks) when evaluating the impact of microarchitectural changes on performance.

This is especially valuable in the context of the increasing necessity to introduce hardware restrictions to reduce power consumption: With conventional methods it is difficult to judge whether a performance loss due to such a restriction (on a simulator) is mostly caused by insufficient scheduling heuristics, which can still be improved to deal better with it, or whether it is largely *inherent* to the restriction.

Not only the influence of microarchitectural parameters, but also of compiler settings can be empirically analyzed with higher validity. For example, the method could be used to study the impact of profiling and alias information—or a lack thereof—on the performance potential of scheduling. When employing scheduling heuristics, the impact depends on how they make use of this information—in contrast, ILP-based analysis allow to abstract away from them. Only the optimization goal has to be clearly defined, which can be done in a comparatively easy and flexible way in this search-based approach: The researcher can formulate different goals in the objective function without caring about how to achieve them.

A further application arises from the fact that we have proven the *correctness* of the basic model. It follows that a schedule is proven to be correct if it is a feasible solution of the ILP (which can be checked in time that is linear in the size of the ILP). This property can be used to *validate* the schedules produced by heuristics. This is a further, inherent advantage of this approach, which does not build on an algorithm, but on a precise mathematical model.

All these applications are not tied to the Itanium Processor Family, but transferable to other architectures that rely on static scheduling. To tap the full potential of the ILP scheduler, it is recommended to integrate it directly into the backend of a compiler: Only this enables access to *high-level information* that is crucial to scheduling and the use of speculation, like (probabilistic) aliasing information.

A different question is whether on the long run ILP-based methods will be fast enough to be integrated into product compilers, or even into dynamic compilation and optimization infrastructures. We will leave this question open here, but notice three trends in support of an affirmative answer:

Firstly, there are vast improvements in the solving performance of ILP solvers (about 40% per year) [BFG$^+$00]. The theory of integer programming is a highly active area of research, which will continue to deliver advances in solution methods—or even fundamental breakthroughs, although the worst-case solution time will likely remain exponential.

Secondly, we can expect that the structure of the developed ILP model has still room for improvements. Possible directions for this have been indicated: Regarding the polytope of the basic model, information on conditional sets of instructions and on minimal distances due to data dependences can be exploited to tighten the precedence constraints (see Sec. 7.1.3) and the resource constraints (see Sec. 8.2.2.1). Among the extensions, especially cyclic code motion seems to harbor further opportunities to improve the structure of the model. Further increases in efficiency provide the necessary headroom for future extensions of the model, such as the global propagation of long latencies (outlined in Sec. 6.7).

Thirdly, computing power continues to improve, not only raw single-task CPU performance, but also and in particular parallel computing power from multi-core chips and multithreading. ILP solving can take advantage of this trend since branch-and-bound algorithms are well parallelizable.

Appendix A

Figures

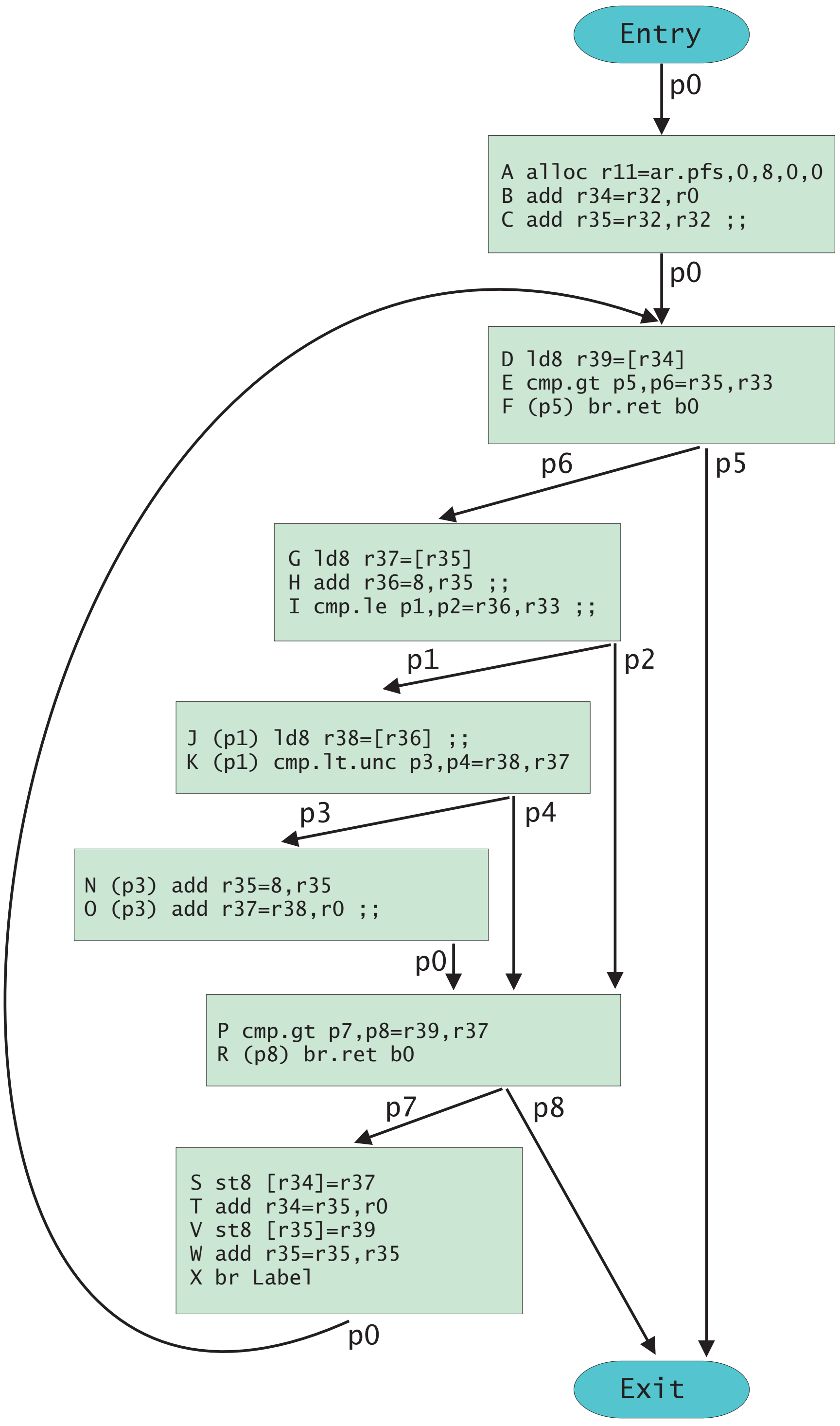


Figure A.1: Basic block graph for Alg. 2; instructions are shown in their respective source blocks.

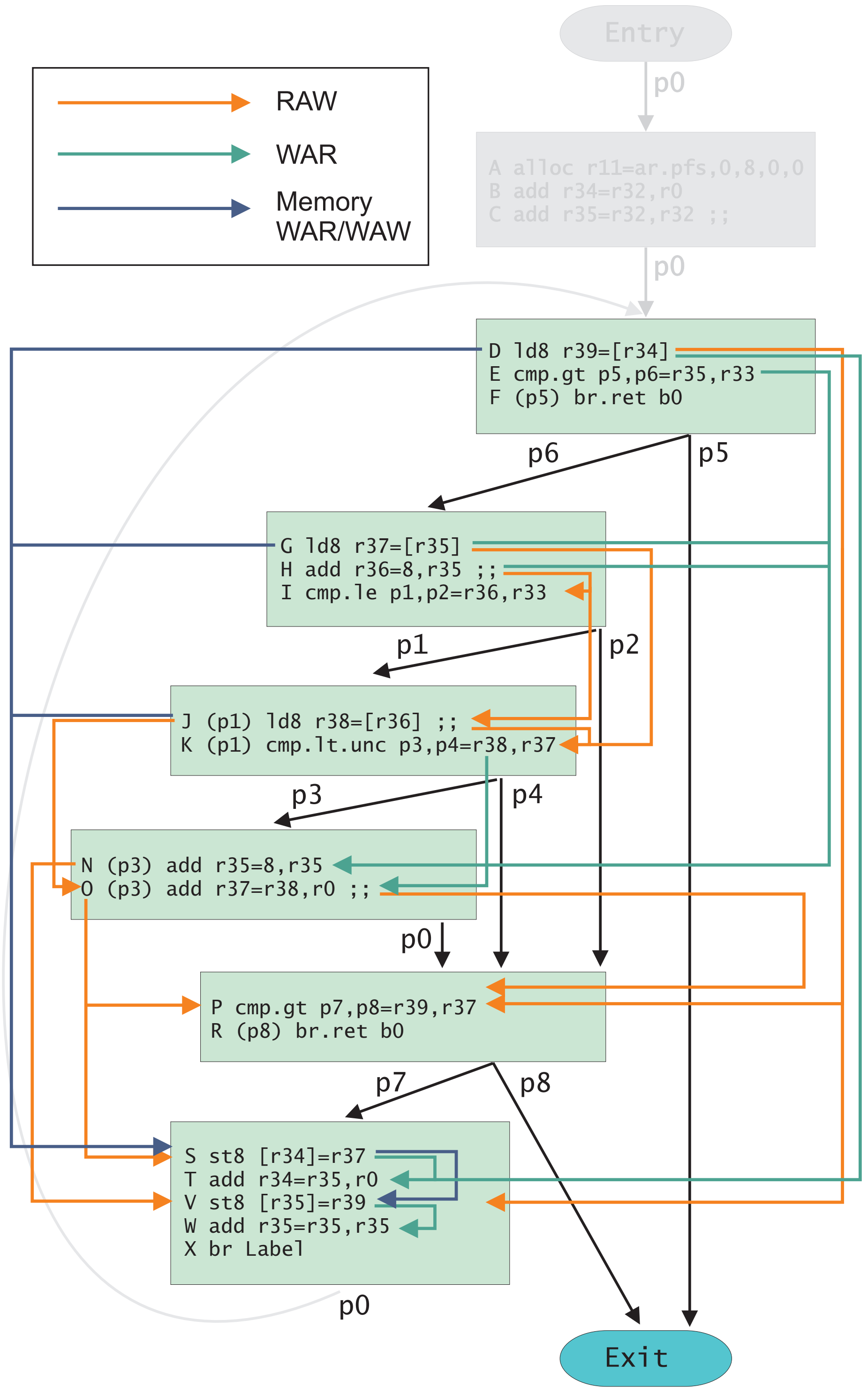


Figure A.2: Data dependence graph for an acyclic subregion of Alg. 2.

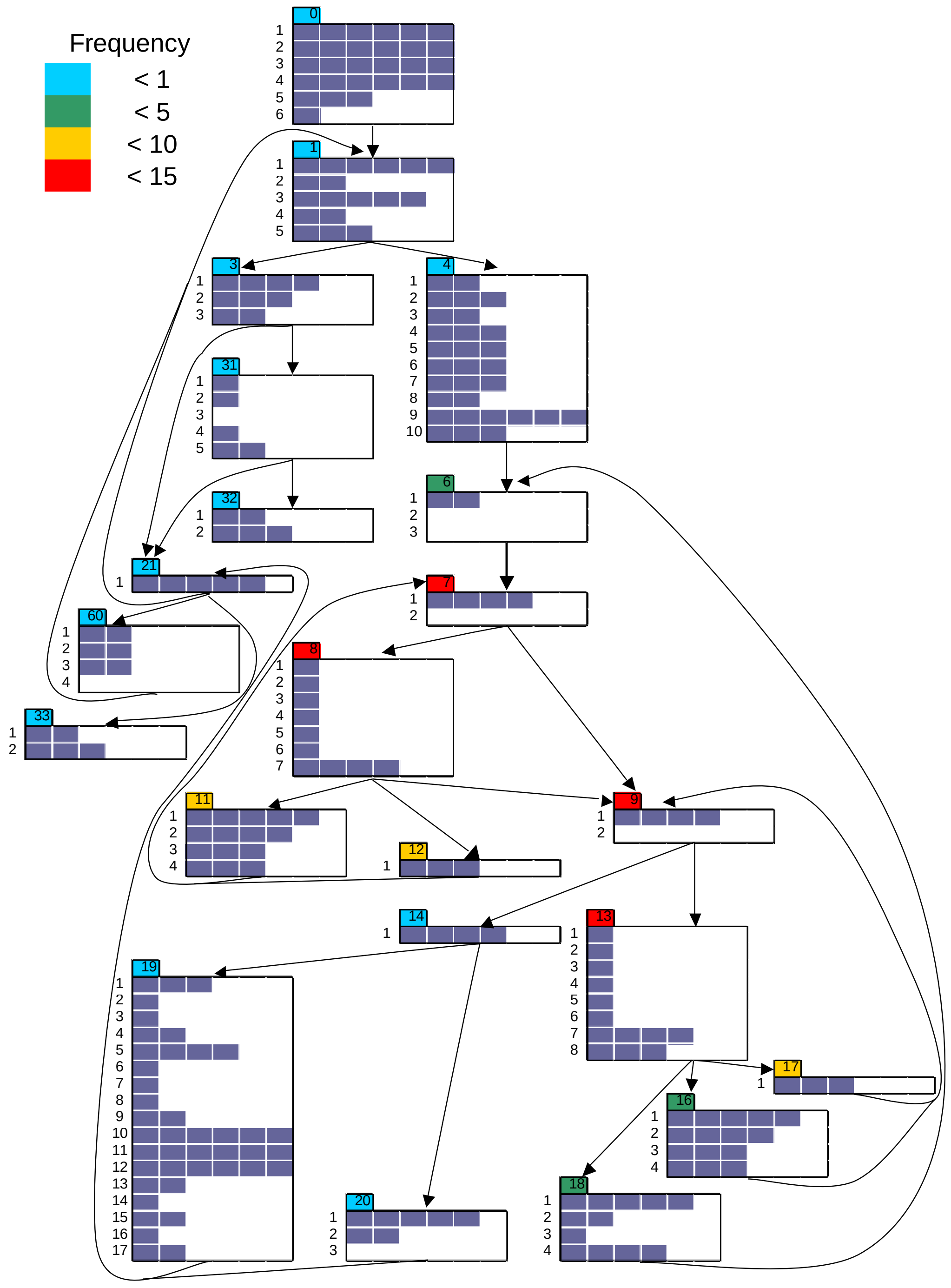


Figure A.3: Input schedule of *qSort3*.

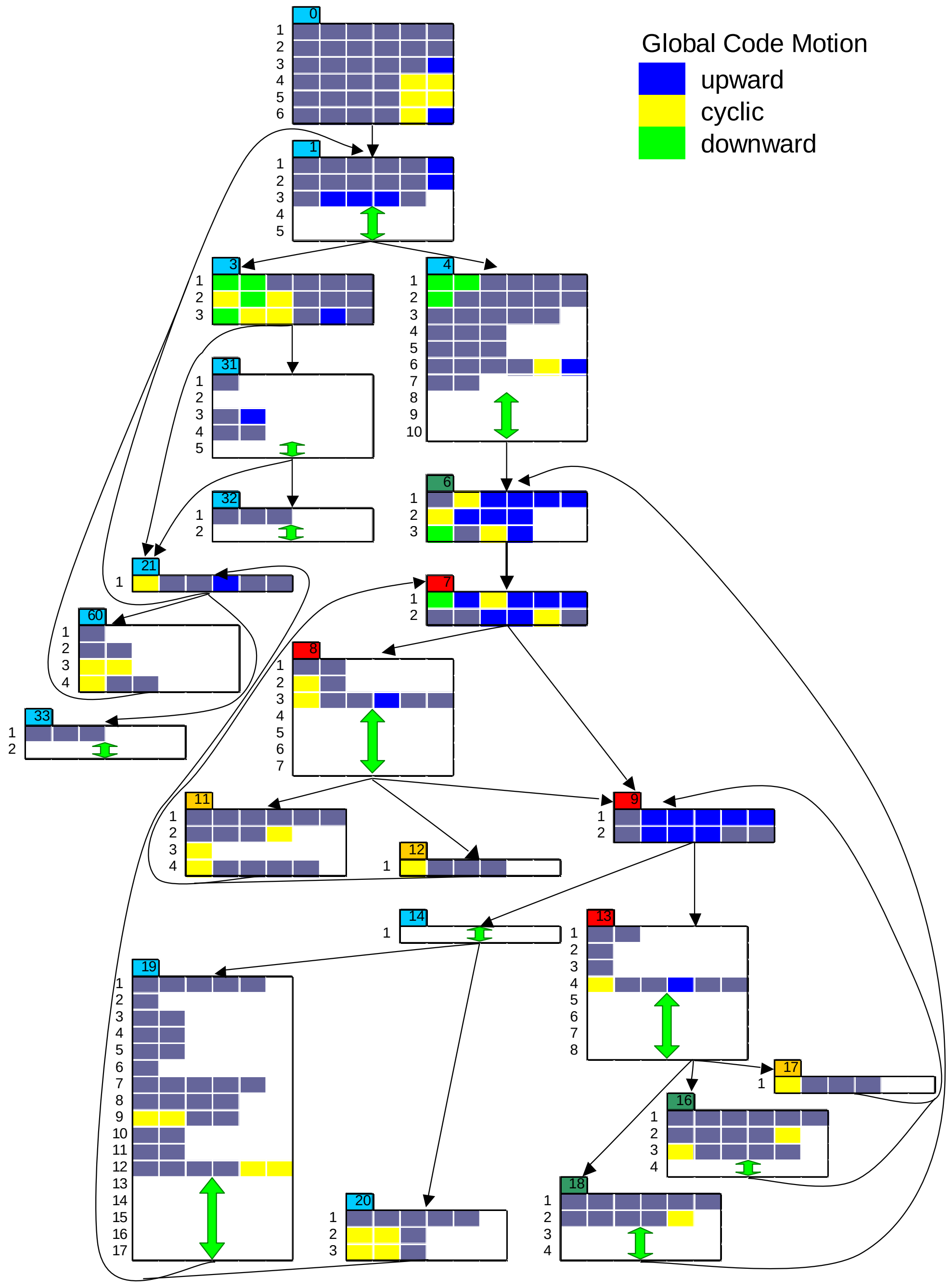


Figure A.4: Output schedule with different kinds of employed code motion.

Appendix B

Proofs

B.1 Proofs for Chapter 4

B.1.1 Proof of Lemma 4.3.11

Lemma *Let* $P = \left\{ x \in \mathbb{R}^n_+ \,\middle|\, Ax \leq b \right\}$ *be an integral polyhedron. If the row indices* $I = \{1, \ldots, m\}$ *of* $A \in \mathbb{R}^{m \times n}$ *are partitioned into two subsets* I_1 *and* I_2*, then*

$$P' = \left\{ x \in \mathbb{R}^n_+ \,\middle|\, A_{I_1} x \leq b_{I_1} \wedge A_{I_2} x = b_{I_2} \right\}$$

is integral (if nonempty). □

PROOF Firstly, note that we can write P' as $\left\{ x \in \mathbb{R}^n_+ \,\middle|\, \hat{A} x \leq \hat{b} \right\}$ with

$$\hat{A} = \begin{pmatrix} A_{I_1} \\ A_{I_2} \\ -A_{I_2} \end{pmatrix} \text{ and } \hat{b} = \begin{pmatrix} b_{I_1} \\ b_{I_2} \\ -b_{I_2} \end{pmatrix}.$$

It is sufficient to show that every extreme point of P' is also an extreme point of P and thus integral. From Theorem 4.1.2 we know that a point in a polyhedron with dimension n is an extreme point if and only if it is the intersection point of n facets of the polyhedron whose corresponding matrix rows are linearly independent.

Let x be an extreme point of P', and let the corresponding n linearly independent matrix rows of $\hat{A}$ be given by a subset $J \subseteq \{1, \ldots, m + |I_2|\}$ such that $\hat{A}_J x = \hat{b}_J$. We note that if $j \in J$ and $j > m$, i.e., if j denotes one of the newly added rows, then $j - |I_2|$ must not be included in J because both rows are linearly dependent: $A_j = (-1) A_{j-|I_2|}$. Let J' be the set where each such j is replaced by $j - |I_2|$, then $\hat{A}_{J'}$ remains linearly independent and $\hat{A}_{J'} x = \hat{b}_{J'}$. Since $\hat{A}_{J'} = A_{J'}$ and $\hat{b}_{J'} = b_{J'}$ this is equivalent to $A_{J'} x = b_{J'}$. Hence $x \in P' \subseteq P$ is also an extreme point of P and thus integral. ■

B.1.2 Proof of Corollary 4.3.12

Corollary *Let $P = \left\{ x \in \mathbb{R}^n_+ \middle| Ax \leq b \right\}$ be an integral polyhedron. If some of the columns of A are removed from the formulation with their corresponding variables, then the resulting polyhedron remains integral (if nonempty).* □

PROOF It is sufficient to consider the case where only one column is being removed: Let A' be the matrix A with the i-th column removed. It remains to be shown that $P'' = \left\{ y \in \mathbb{R}^{n-1}_+ \middle| A'y \leq b \right\}$ (assumed nonempty) is integral.

Since $x \in \mathbb{R}^n_+$, we can write P as $\left\{ x \in \mathbb{R}^n_+ \middle| Ax \leq b \wedge -x_i \leq 0 \right\}$, and with the preceding Lemma 4.3.11 we can conclude that the polytope $P' = \left\{ x \in \mathbb{R}^n_+ \middle| Ax \leq b \wedge -x_i = 0 \right\}$ is integral, too. Then from Def. 4.1.1 it is clear that for any extreme point $z = (z_1, \ldots, z_{n-1})$ of $P'' = \left\{ x \in \mathbb{R}^{n-1}_+ \middle| A'x \leq b \right\}$ the point $(z_1, \ldots, z_{i-1}, 0, z_i, \ldots, z_{n-1})$ is an extreme point of P' and thereby integral. It follows that also z must have been integral, and hence the polytope P'', too. ■

B.1.3 Proof of Lemma 4.3.13

Lemma *If $P = \left\{ x \in \mathbb{R}^n_+ \middle| Ax \leq b \right\}$ is an integral polytope, then so is $proj_J(P)$.* □

PROOF Since $\forall k \in J : proj_J(S) = proj_{J\setminus\{k\}}(proj_{\{k\}}(S))$, it is sufficient to consider a projection onto a $(\dim(P) - 1)$-dimensional subspace, i.e., to prove the lemma for a set $J = \{k\}$.

The projection of a polyhedron is a polyhedron according to [NW88], yet it remains to be proven that the polytope $proj_J(P)$ is integral. For this, we show in the remainder of this proof that for each extreme point y of $proj_J(P)$, there exists an extreme point x of P such that $proj_J(x) = y$. Then since x is integral, $proj_J(x) = y$ is integral, too. It follows the integrality of $proj_J(P)$.

We assume the opposite: Let y be an extreme point of $proj_J(P)$, and let all the points $Q = \{x \in P | proj_J(x) = y\}$ *not* be extreme points. Then also this vector $x \in Q$ that is *minimal* with respect to the k-th component is not an extreme point, i.e., there exist $w, y \in P, w \neq y$ and a $c \in]0, 1[$ such that $cw + (1 - c)y = x$. However, due to the minimality, w and y must differ on other components than the k-th (otherwise either w_k or y_k had to be less than x_k). It follows $proj_J(w) \neq proj_J(y)$, and since these two points are in $proj_J(P)$ and $y = c \cdot proj_J(w) + (1 - c) \cdot proj_J(y)$, y cannot be an extreme point. This contradiction concludes the proof. ■

B.1.4 Proof of Lemma 4.3.14

Lemma *Let $P = \left\{ x \in \mathbb{R}^n_+ \middle| Ax \leq b \right\}$ be an integral polyhedron and let the $m \times n'$ matrix A' be the matrix A with some columns duplicated. Then $P' = \left\{ x \in \mathbb{R}^{n'}_+ \middle| A'x \leq b \right\}$ is integral.* □

PROOF Let A' be the matrix A with the last column duplicated. Let there be an arbitrary extreme point y of $P' = \left\{ x \in \mathbb{R}^{n+1}_+ \middle| A'x \leq b \right\}$ given; it is sufficient to show that y is integral. According to Theorem 4.1.2, there exist $n+1$ linearly independent matrix rows of A', given by a subset $J \subseteq \{1, \ldots, n+1\}$, such that $A'_J y = b_J$.

As mentioned in Def. 4.1.3, we can assume that A' contains the inequalities $x_i \geq 0$ ($\forall 1 \leq i \leq n+1$), i.e., we can assume that the constraint $x_{n+1} \geq 0$ for the new variable x_{n+1} is automatically added when the new column is appended. Then either $x_n \geq 0$ or $x_{n+1} \geq 0$ must be included in the above selection of $n+1$ constraints that are satisfied with equality—otherwise the last two columns in A'_J were equal, so that the column rank of A'_J had to be less than $n+1$. Since the row rank is equal to the column rank, this would contradict the fact that A'_J has $n+1$ linearly independent rows and thus full rank.

Hence either $y_n = 0$ or $y_{n+1} = 0$—in the first case we remove from A'_J the n-th column and the row belonging to $x_n \geq 0$, in the second case the ($n+1$)-th column and the row belonging to $x_{n+1} \geq 0$. The resulting $n \times n$ matrix then is of the form $A_{J'}$ for a subset $J' \subseteq \{1, \ldots, n\}$ and still has full rank; in addition, either $y' = (y_1, \ldots, y_{n-1}, y_{n+1})$ or $y' = (y_1, \ldots, y_n)$ is the unique solution of $A_{J'} y' = b_{J'}$. Hence y' is an extreme point of P (with the backward direction of Theorem 4.1.2) and thus integral. Then y must be integral, too. ■

B.2 Proofs for Section 5.1

B.2.1 Proof of Theorem 5.1.17

Theorem *Any of the constraints (5.1.8) for a maximal path with $\tau_2(P) \geq 2$ is a linear combination of those constraints with $\tau_2(P) < 2$.* □

PROOF We first observe that each maximal path in $G_s[N(C)]$ (for a given a control flow path $C \in \mathcal{C}$) is uniquely characterized by the order of its type 2 edges. Concretely, let a sequence of type 2 edges be given as $(e_1, \ldots, e_s)$. In the following, we denote the components of a type 2 edge e_i always as (u_i, v_{i+1}), and the instruction of node u_i as n_i. Then the subpath from v_i to u_i is unique; it consists of all the nodes in $N(C) \cap N(n_i)$ between u_i and v_i. If $N(C) \cap N(n_i)$ is abbreviated by N_i, then the left-hand side of constraints (5.1.8) can be written as:

$$\Gamma(e_1, \ldots, e_s) := \sum_{\substack{a \in N_1 \\ a \succeq u_1}} x_a + \sum_{i=2}^{s} \sum_{\substack{a \in N_i \\ u_i \preceq a \preceq v_i}} x_a + \sum_{\substack{a \in N_{s+1} \\ a \preceq v_{s+1}}} x_a$$

Now we prove the claim. Our goal is to show that, given a control flow path $C \in \mathcal{C}$ and a maximal path $P \subseteq G_s[N(C)]$ with two or more type 2 edges $e_1, \ldots, e_t$, the inequality $1 \geq \Gamma(e_1, \ldots, e_t)$ can be written as a linear combination of those constraints (5.1.8) representing paths with either *one or zero* type 2 edges.

For all $1 \leq i \leq t$, instances of the former constraint sort are given by $\Gamma(e_i) \leq 1$. They correspond to maximal paths with exactly one type 2 edge e_i. Instances that represent maximal paths with zero type 2 edges are given by $\sum_{a \in N_i} x_a = 1$ ($\forall 1 \leq i \leq t$) (see Theorem 5.1.15). To deduce $1 \geq \Gamma(e_1, \ldots, e_t)$, we first add up t and $t-1$ instances of these constraints, respectively:

$$t \;\geq\; \sum_{i=1}^{t} \Gamma(e_i) \tag{B.2.1}$$

$$t-1 \;=\; \sum_{i=2}^{t} \sum_{a \in N_i} x_a \tag{B.2.2}$$

Subtracting (B.2.2) from (B.2.1) then yields:

$$\begin{aligned}
1 &\geq \sum_{i=1}^{t} \Gamma(e_i) - \sum_{i=2}^{t} \sum_{a \in N_i} x_a \\
&= \sum_{i=1}^{t} \left(\sum_{\substack{a \in N_i \\ a \succeq u_i}} x_a + \sum_{\substack{a \in N_{i+1} \\ a \preceq v_{i+1}}} x_a \right) - \sum_{i=2}^{t} \sum_{a \in N_i} x_a \\
&\overset{(*)}{=} \sum_{i=2}^{t} \left(\sum_{\substack{a \in N_i \\ u_i \preceq a \preceq v_i}} x_a + \sum_{a \in N_i} x_a \right) + \sum_{\substack{a \in N_1 \\ a \succeq u_1}} x_a + \sum_{\substack{a \in N_{t+1} \\ a \preceq v_{t+1}}} x_a - \sum_{i=2}^{t} \sum_{a \in N_i} x_a \\
&= \Gamma(e_1, \ldots, e_t)
\end{aligned}$$

In step $(*)$, we have extracted the first addend ($i = 1$) from the first summation notation and then reorganized the latter in such a way that the i-th addend contains only variables x_a such that $a \in N_i$. Previously, these variables were added in consecutive addends. The total of the sum remains unchanged. ■

B.3 Proofs for Section 5.2

B.3.1 Proof of Theorem 5.2.8

Theorem *Let a resource flow network be given as defined previously in Def. 5.2.7. Then the following* network flow resource constraints, *generated* $\forall A \in \mathcal{B}, \forall t \in G(A)$, *form an* integral *polytope of resource constraints:*

$$\sum_{\forall k:(k,l) \in E'} y^{At}_{(k,l)} = \sum_{\forall m:(l,m) \in E'} y^{At}_{(l,m)} \qquad \forall l \in \mathcal{R} \tag{B.3.1}$$

$$\sum_{\forall k:(k,l) \in E'} y^{At}_{(k,l)} \leq c(l) \qquad \forall l \in \mathcal{R} \tag{B.3.2}$$

E' contains all edges from E plus a special edge (s, l) for each $l \in \vec{\mathcal{R}}$ with the sole purpose to model the inflow into this node. For each edge $e \in E$ there exists a new integral flow variable

y_e^{At} *which holds the value of the flow through this edge; in addition, the* inflow variables $y_{(s,l)}^{At}$ *model the inflow into node* l. □

PROOF It is obvious that the constraints model a resource flow network as described in Def. 5.2.7. Regarding the integrality claim, we employ the following theorem to prove that the constraint matrix is totally unimodular. The integrality then follows with Theorem 4.3.5.

Theorem B.3.1 *A matrix* $A \in \mathbb{R}^{m \times n}$ *is totally unimodular if and only if for every* $J \subseteq \{1, \dots, m\}$, *there exists a partition* J_1, J_2 *of* J *such that*

$$\left| \sum_{i \in J_1} a_{ij} - \sum_{i \in J_2} a_{ij} \right| \leq 1 \qquad \forall j = 1, \dots, n$$

□

Let there be a subset $J \subseteq \{1, \dots, m\}$ of the rows of the constraint matrix given. Each row corresponds either to a constraint (B.3.1) or (B.3.2), generated for a node $j \in \mathcal{R}$. For the constraints (B.3.1), we can assume that the variables on the left-hand side of the equation are represented by coefficients 1 in the row and those on the right-hand side by -1.

Let the sets A and B contain the instances of constraints (B.3.1) and (B.3.2) whose left-hand sides are included in the row selection, respectively. Moreover, let $\mathcal{R}_A^J \subseteq \mathcal{R}$ denote the set of all nodes for which the constraints in A were generated and $\mathcal{R}_B^J$ the analogous set for B.

Now we construct the sets J_1 and J_2 with the desired properties: First, we collect all the rows due to (B.3.1) in J_1. Then we make the following observation about the j-th column of these rows (for any $j \in \{1, \dots, n\}$): If this column corresponds to a variable $y_{(p,q)}^{At}$, then the sum $\sum_{i \in J_1} a_{ij}$ has either value

- -1 if $q \notin \mathcal{R}_A^J$ and $p \in \mathcal{R}_A^J$,
- 0 if $q \notin \mathcal{R}_A^J$ and $p \notin \mathcal{R}_A^J$, or if $q \in \mathcal{R}_A^J$ and $p \in \mathcal{R}_A^J$,
- or 1 if $q \in \mathcal{R}_A^J$ and $p \notin \mathcal{R}_A^J$.

Now we use this observation to distribute the rows due to (B.3.2) to the sets J_1 and J_2: Such a row created for an $l \in \mathcal{R}_B^J$ is included in J_1 if $l \notin \mathcal{R}_A^J$ and in J_2 if $l \in \mathcal{R}_A^J$. This construction yields $\sum_{i \in J_1} a_{ij} - \sum_{i \in J_2} a_{ij} \in \{-1, 0, 1\}$ for all $j = 1, \dots, n$ so that the matrix is totally unimodular. ■

B.3.2 Proof of Theorem 5.2.11

Theorem *The value of the maximum flow of a network flow problem with node capacities is exactly the minimal capacity of a complete node cut.* □

PROOF Let a network flow problem be given as described in Def. 5.2.3, but with node capacities. We transform it into a problem with edge capacities by replacing each node with two new nodes connected by a new edge that has the same capacity as the previous node capacity. All other edges are assigned a capacity that is so large that it is never limiting, for instance as large as the total capacity of all nodes in the original problem. Then we can apply the max-flow min-cut theorem 5.2.4 to this graph and transfer the result back to the original problem, based on the following two observations:

1. The value of a maximal flow is equal in both the original and the transformed problem.
2. The minimal capacity of a complete node cut in the original problem equals the minimal capacity of a set of edges to disconnect G with s and t in different components in the transformed problem.

The first observation is evident. To prove (2), it is sufficient to show that for each complete node cut in the original problem there exists an edge set as described in (2) *with the same capacity* and vice versa. The *forward implication* of this is clear since each such node cut corresponds to a set of edges in the transformed problem that has the same capacity and disconnects the source and the sink as required by Theorem 5.2.4 (this set consists of exactly those newly introduced edges that correspond to the nodes in the node cut).

For the *backward direction*, let a disconnecting set of edges with minimal capacity be given—this set then can only consist of the newly created edges. The nodes in the original problem corresponding to these edges must constitute a complete node cut according to Lemma 5.2.10 (2) $\Rightarrow$ (1) (with the same capacity).

This proven correspondence, expressed by the two observations, allows to transfer the edge capacities version of the max-flow min-cut theorem to the node capacities version, which concludes the proof. ■

B.3.3 Proof of Theorem 5.2.19

In the following, h_S refers to an instance of the inflow constraints (5.2.8) generated for a node cut S. $\mathbb{P}^{-1}(S)$ refers to those nodes that are postdominated by S, but not element of S:

$$\mathbb{P}^{-1}(S) := \mathbb{P}_+^{-1}(S) \setminus S$$

Further, we frequently use notation from Sec. 1.3.1 like $V^{\succ}(S)$ and $\mathcal{C}^{\succeq}(x)$; we write $P \cap S \neq \emptyset$ if a path $P \in \mathcal{C}^{\succeq}(x)$ does not traverse a block from S. We first prove a lemma and a proposition needed in the proof of the theorem:

Lemma B.3.2 *For each subset $S \subseteq V$ there exists a linear combination of basic constraints (5.2.8) that subsumes*

$$\sum_{v \in \mathbb{P}_+^{-1}(S)} y_{(s,v)}^{At} \leq c(S) \tag{B.3.3}$$

□

PROOF We assume that S is not a node cut, otherwise the lemma follows with Corollary 5.2.17. We remove successively all nodes from S that are dominated or postdominated by the others until we have reached a subset that is a node cut. This node cut is below included in a linear combination with the required properties.

If a node $v \in S$ is dominated or postdominated by $S \setminus \{v\}$, then $\mathbb{P}^{-1}(S) \subseteq \mathbb{P}^{-1}(S \setminus \{v\})$ since every path from a node in $\mathbb{P}^{-1}(S)$ to v still intersects $S \setminus \{v\}$. Thus the removal of this node from S does not change the set $\mathbb{P}^{-1}(S)$. However, v is possibly removed from $\mathbb{P}_+^{-1}(S) = S \cup \mathbb{P}^{-1}(S)$.

Thus we have $\mathbb{P}_+^{-1}(S)$=$\mathbb{P}_+^{-1}(S \setminus \{v\}) \cup \{v\}$. Let $D = \{d_1, \ldots, d_k\}$ be the nodes removed this way and $S' := S \setminus D$ the resulting node cut, then it follows $\mathbb{P}_+^{-1}(S) = \mathbb{P}_+^{-1}(S') \cup D$.

Now we conclude the proof by showing that the *sum* of the constraints $h_{S'}$ and $h_{\{d_1\}}, \ldots, h_{\{d_k\}}$ subsumes (B.3.3)—the lemma follows then with Corollary 5.2.17:

Firstly, from $\mathbb{P}_+^{-1}(S) = \mathbb{P}_+^{-1}(S') \cup D$ it follows that each variable on the left-hand side of (B.3.3) occurs also on the left-hand side of this sum (with coefficient ≥ 1). Secondly, since $c(S) \overset{S \supseteq D}{=} c(S \setminus D) + c(D) = c(S') + c(d_1) + \ldots + c(d_k)$, the right-hand sides are equal. Hence the sum subsumes (B.3.3). ∎

Proposition B.3.3 $\mathbb{P}_+^{-1}(A) \subseteq \mathbb{P}_+^{-1}(B) \Rightarrow \mathbb{P}_+^{-1}(C) \subseteq \mathbb{P}_+^{-1}(C \setminus A \cup B)$

PROOF Let $\mathbb{P}_+^{-1}(A) \subseteq \mathbb{P}_+^{-1}(B)$ and let there be an $x \in \mathbb{P}_+^{-1}(C)$. Every path P in $\mathcal{C}^{\succeq}(x)$ passes through C—either through $C \cap A$ or through $C \setminus A \subseteq C \setminus A \cup B$. In the former case it also intersects $B \subseteq C \setminus A \cup B$ since $A \subseteq \mathbb{P}_+^{-1}(B)$. Hence P intersects $C \setminus A \cup B$ in both cases, thus $x \in \mathbb{P}_+^{-1}(C \setminus A \cup B)$. ∎

Now we are ready to prove Theorem 5.2.19:

Theorem *All basic inflow constraints constitute a* minimal *description of the inflow polytope.* □

PROOF Let S be an atomic and tight node cut. It is sufficient to show that there exists no linear combination of other basic constraints that subsumes h_S (using Corollary 5.2.17). The proof is by contradiction: we assume that such a linear combination exists and is given by atomic and tight node cuts $S_1, \ldots, S_k$ and a $\lambda \in \mathbb{R}_+^k$. Subsumption is a property of the coefficient matrix, as expressed by Lemma 4.3.2: Firstly, since for all $v \in \mathbb{P}_+^{-1}(S)$ the coefficient of $y_{(s,v)}^{At}$ in h_S is one, the same coefficient in the linear combination must be greater or equal to one:

$$\sum_{\forall i : v \in \mathbb{P}_+^{-1}(S_i)} \lambda_i \geq 1 \qquad \forall v \in \mathbb{P}_+^{-1}(S) \tag{B.3.4}$$

$$\hat{c} := \sum_{i=1}^{k} \lambda_i c(S_i) \leq c(S) \tag{B.3.5}$$

Secondly, Equ. (B.3.5) states that the right-hand side of the linear combination, its “cost” $\hat{c}$, must not be greater than $c(S)$. We assume here that the given linear combination is *minimal* with respect to this cost. Note that from this minimality it follows $\forall i : \lambda_i \leq 1$ and that each S_i is a subset of $V^{\prec}(S) \cup S \cup V^{\succ}(S)$—otherwise we could intersect each S_i with this set and the result would still satisfy (B.3.4), but have lower cost. Using this observation, we will conduct the proof in three parts:

1. The first shows $\forall i : S_i \cap V^{\succ}(S) = \emptyset$,
2. the second $\forall i : S_i \cap V^{\prec}(S) = \emptyset$, and
3. the third will finally derive a contradiction from the remaining possibility $\forall i : S_i \subseteq S$.

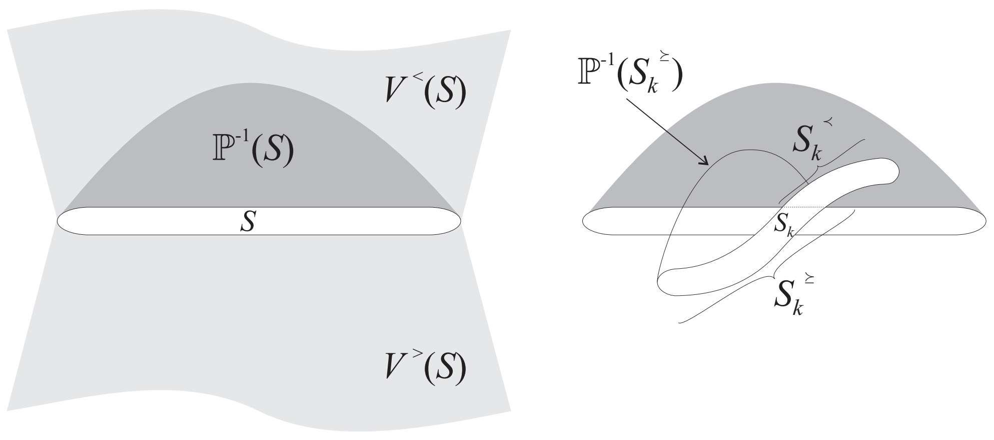
Part 1:

Let there be an S_k such that $S_k \cap V^{\succ}(S) \neq \emptyset$. We define

$$S_k^{\succeq} := S_k \cap V^{\succeq}(S) \qquad S_k^{\succ} := S_k \cap V^{\succ}(S)$$

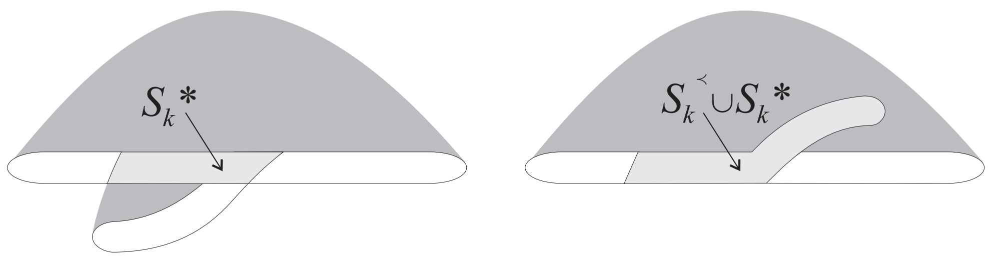
$$S_k^{\preceq} := S_k \cap V^{\preceq}(S) \qquad S_k^{\prec} := S_k \cap V^{\prec}(S)$$

and $S_k^* := \mathbb{P}_+^{-1}(S_k^{\succeq}) \cap S$ (see the schematic illustrations in Figure B.1 and Figure B.2 (a)). We will derive a contradiction from each of the following two cases: either $c(S_k^{\succeq}) \leq c(S_k^*)$ or $c(S_k^{\succeq}) > c(S_k^*)$.



(a) S and related node sets. (b) Illustration of S_k.

Figure B.1: Illustration of Part 1



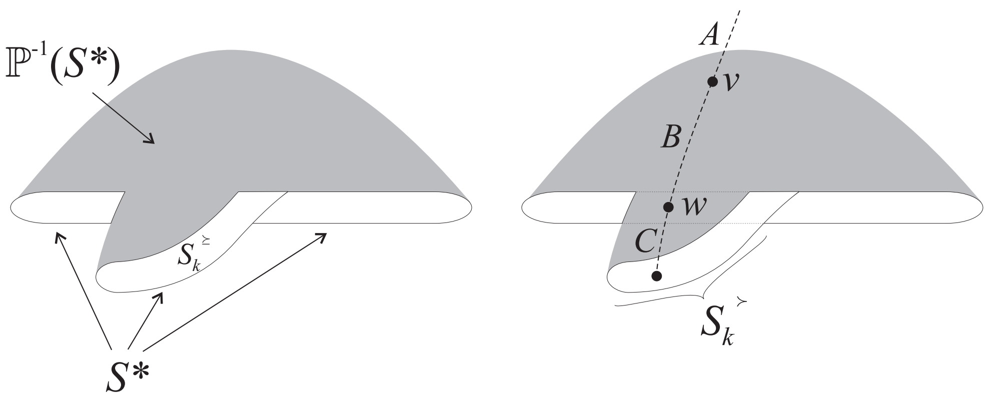
(a) Illustration of S_k^*. (b) $S_k^{\prec} \cup S_k^*$ from Part 1, Second Case.

Figure B.2: The node sets S_k^* and $S_k^{\prec} \cup S_k^*$.

First Case

$c(S_k^{\succeq}) \leq c(S_k^*)$: Then we can remove all blocks from S that are element of S_k^* and replace them by the blocks in $S_k^{\succeq}$. We show in the following that this delivers a node cut that subsumes S, but has lower cost, which delivers a contradiction with the assumption that S is tight (see Theorem 5.2.16-(2)).

From the definitions of S_k^* and $S_k^{\succeq}$ it follows $\mathbb{P}_+^{-1}(S_k^*) \subseteq \mathbb{P}_+^{-1}(S_k^{\succeq})$ and with Proposition B.3.3 $\mathbb{P}_+^{-1}(S) \subseteq \mathbb{P}_+^{-1}(S \setminus S_k^* \cup S_k^{\succeq})$. We define $S^* := S \setminus S_k^* \cup S_k^{\succeq}$—as illustrated in Fig. B.3 (a)—and remove in the following successively all nodes from S^* that are dominated or postdominated by the others until $S^* \subseteq S \setminus S_k^* \cup S_k^{\succeq}$ is a node cut. The latter then is shown to be "tighter" than S.



(a) Illustration of S^*.

(b) The path from Part 1, First Case.

Figure B.3: Illustration of Part 1, First Case

We first remove all nodes that are **dominated** by others: Let us assume that a node $v \in S^*$ is removed because it is dominated by a set $D \subseteq S^* \setminus \{v\}$. Then $v \notin S$, since $v \in S$ would imply $D \subseteq S$, which contradicts the assumption that S is a node cut (by Def. 5.2.9).

It follows $v \in S_k^{\succ}$. Hence only nodes from $S_k^{\succ} = S^* \cap V^{\succ}(S)$ are removed, and since the removal does not change $\mathbb{P}^{-1}(S^*)$ (see the proof of Lemma B.3.2), it still holds $\mathbb{P}_+^{-1}(S) \subseteq \mathbb{P}_+^{-1}(S^*)$ after the removal. We can also prove the stronger result $\mathbb{P}_+^{-1}(S) \subset \mathbb{P}_+^{-1}(S^*)$ by showing that not *all* nodes from $S_k^{\succ}$ are removed this way:

Since S_k is atomic, there must exist a $v \in \mathbb{P}_+^{-1}(S_k)$ such that $v \notin \mathbb{P}_+^{-1}(S_k^{\preceq})$ and $v \notin \mathbb{P}_+^{-1}(S_k^{\succ})$ (Theorem 5.2.16-(1)). It follows that there must exist a path from v to a node in $S_k^{\preceq}$ (otherwise $v \in \mathbb{P}_+^{-1}(S_k^{\succ})$), hence $v \in V^{\preceq}(S)$, and that there must exist a path P from v to a node in $S_k^{\succ}$ that does not intersect $S_k^{\preceq}$ (otherwise $v \in \mathbb{P}_+^{-1}(S_k^{\preceq})$). But P intersects S (since $v \in V^{\preceq}(S)$)—let $w \in S \cap P$ be the *first* node on the path that is in S. We make three observations:

- From the construction of P we have $w \notin S_k^{\preceq}$ and $w \notin S_k^{\succeq}$. $w \in \mathbb{P}_+^{-1}(S_k^{\succeq})$ holds since the subpath from v to w (denoted by B in the following) does not intersect S_k before w although $v \in \mathbb{P}_+^{-1}(S_k)$; thus any path going out from w must intersect $S_k^{\succeq}$. It follows $w \in S_k^* \setminus S_k^{\succeq} \Rightarrow w \notin S^*$, so that B has no node in common with S^*.

- Since either $v \in S$ or $v \in V^{\prec}(S)$, v is not dominated by other nodes from S or from S^*. Thus we can obtain a path A from an entry node to v that does not intersect S^*.
- Since $w \in S$ is not postdominated by other nodes in S and $w \in \mathbb{P}_+^{-1}(S_k^{\succeq})$, there exists a path C from w to a node in $S_k^{\succ}$ that does not intersect S^* before this node.

The concatenation of A, B, and C is a path that starts at an entry node and does not intersect S^* before it reaches a node in $S_k^{\succ}$ (see Figure B.3 (b)): this node then cannot be dominated by other nodes of S^* since the path contains none of them. Thus after the removal of all nodes dominated by others from S^* it still holds:

$$S^* \cap S_k^{\succ} \neq \emptyset$$

So there is at least one element in S^* that is not in S and its predecessors, hence $\mathbb{P}_+^{-1}(S) \subset \mathbb{P}_+^{-1}(S^*)$.
Removing a node from S^* that is **postdominated** by the others does not change $\mathbb{P}_+^{-1}(S^*)$. Thus it still holds $\mathbb{P}_+^{-1}(S) \subset \mathbb{P}_+^{-1}(S^*)$ after all these nodes have been removed, and S^* is then by definition a node cut. Together with

$$c(S^*) \leq c(S \setminus S_k^* \cup S_k^{\succeq}) \overset{S_k^* \subseteq S}{\leq} c(S) - c(S_k^*) + c(S_k^{\succeq}) \leq c(S)$$

this contradicts the assumption that S is tight (as defined in Theorem 5.2.16-(2)), and refutes with this the first case.

Second Case

$c(S_k^{\succeq}) > c(S_k^*)$: Then we can remove all blocks from S_k that are element of $S_k^{\succeq}$ and replace them by the blocks in S_k^*. This can be used to reduce the cost of the given linear combination of basic constraints (which includes h_{S_k}), contradicting the assumption that it had already minimal cost.
We first show $\mathbb{P}_+^{-1}(S) \cap \mathbb{P}_+^{-1}(S_k^{\prec} \cup S_k^{\succeq}) = \mathbb{P}_+^{-1}(S) \cap \mathbb{P}_+^{-1}(S_k^{\prec} \cup S_k^*)$:

$$\begin{aligned}
& x \in \mathbb{P}_+^{-1}(S) \cap \mathbb{P}_+^{-1}(S_k^{\prec} \cup S_k^{\succeq}) \\
\Leftrightarrow\ & \forall P \in \mathcal{C}^{\succeq}(x) : P \cap S \neq \emptyset \wedge P \cap \left(S_k^{\prec} \cup S_k^{\succeq}\right) \neq \emptyset \\
\Leftrightarrow\ & \forall P \in \mathcal{C}^{\succeq}(x) : (P \cap S \neq \emptyset \wedge P \cap S_k^{\prec} \neq \emptyset) \vee \left(P \cap S \neq \emptyset \wedge P \cap S_k^{\succeq} \neq \emptyset\right) \\
\overset{(*)}{\Leftrightarrow}\ & \forall P \in \mathcal{C}^{\succeq}(x) : (P \cap S \neq \emptyset \wedge P \cap S_k^{\prec} \neq \emptyset) \vee (P \cap S \neq \emptyset \wedge P \cap S_k^* \neq \emptyset) \\
\Leftrightarrow\ & x \in \mathbb{P}_+^{-1}(S) \cap \mathbb{P}_+^{-1}(S_k^{\prec} \cup S_k^*)
\end{aligned}$$

The equivalence $(*)$ is the pivotal step and requires further elaboration:
"$\Rightarrow$": Let $P \in \mathcal{C}^{\succeq}(x)$ be a path such that $P \cap S \neq \emptyset \wedge P \cap S_k^{\succeq} \neq \emptyset$ and let $P \cap S_k^{\prec} = \emptyset$ (otherwise the implication is trivial). Furthermore, let $v \in P \cap S$ be the first node on P that is in S. Then $v \in \mathbb{P}_+^{-1}(S_k^{\succeq})$—otherwise there would be a path $Q \in \mathcal{C}^{\succeq}(v)$ that does not intersect $S_k^{\succeq}$ (and $S_k^{\prec}$). The concatenation of P between x and v and Q would then not intersect S_k although it is element of $\mathcal{C}^{\succeq}(x)$, which contradicts $x \in \mathbb{P}_+^{-1}(S_k)$.
From $v \in \mathbb{P}_+^{-1}(S_k^{\succeq}) \cap S = S_k^*$ it follows then $P \cap S_k^* \neq \emptyset$.

"$\Leftarrow$": Follows from $S_k^* \subseteq \mathbb{P}_+^{-1}(S_k^{\succeq})$.

The proven equation $\mathbb{P}_+^{-1}(S) \cap \mathbb{P}_+^{-1}(S_k) = \mathbb{P}_+^{-1}(S) \cap \mathbb{P}_+^{-1}(S_k^{\prec} \cup S_k^*)$ together with

$$c(S_k) = c(S_k^{\prec} \cup S_k^{\succeq}) > c(S_k^{\prec} \cup S_k^*)$$

shows that it is possible to replace h_{S_k} by $h_{S_k^{\prec} \cup S_k^*}$ in the linear combination without violating (B.3.4) and (B.3.5) (both sets are illustrated in Fig. B.1 (b) and Fig. B.2 (b), respectively). The new linear combination then even has strictly lower cost (the left-hand side of (B.3.5)). However, in order to constitute a contradiction with the assumption that the linear combination had minimal cost before, the new linear combination may only use basic constraints and not h_S itself.

$h_{S_k^{\prec} \cup S_k^*}$ is not necessarily a basic constraint. But from Lemma B.3.2 we know that there exists a linear combination of basic constraints that subsumes it. If h_{S_k} is replaced by this linear combination, then the resulting whole linear combination—denoted by the node cuts $\hat{S}_1, \ldots, \hat{S}_l$ and a $\lambda \in \mathbb{R}_+^l$ in the following—still subsumes h_S and still has a lower cost than $\hat{c}$.

It remains the possibility that this linear combination contains h_S itself: Without restricting generality let us assume that $\hat{S}_l = S$. Then $\lambda_l < 1$ (since the linear combination has lower cost than $\hat{c}$) and we can omit $h_{\hat{S}_l}$ and instead divide all λ_i, $i = 1, \ldots, l-1$, by $1 - \lambda_l$. The linear combination modified this way still satisfies (B.3.4):

$$\forall v \in \mathbb{P}_+^{-1}(S) : \sum_{\substack{\forall i : v \in \mathbb{P}_+^{-1}(\hat{S}_i) \\ i \neq l}} \left(\frac{1}{1-\lambda_l}\right) \lambda_i = \frac{1}{1-\lambda_l} \left(\underbrace{\sum_{\forall i : v \in \mathbb{P}_+^{-1}(\hat{S}_i)} \lambda_i}_{\geq 1} - \lambda_l \right) \geq \frac{1}{1-\lambda_l}(1-\lambda_l) = 1$$

But it has a lower cost than $\hat{c}$:

$$\begin{aligned} \sum_{i=1}^{l-1} \left(\frac{1}{1-\lambda_l}\right) \lambda_i c(\hat{S}_i) &= \frac{1}{1-\lambda_l} \left(\underbrace{\sum_{\forall i : v \in \mathbb{P}_+^{-1}(\hat{S}_i)} \lambda_i c(\hat{S}_i)}_{< \hat{c}} - \lambda_l c(\hat{S}_l) \right) \\ &\overset{\hat{S}_l = S}{<} \frac{\hat{c} - \lambda_l c(S)}{1-\lambda_l} \overset{\hat{c} \leq c(S)}{\leq} \frac{\hat{c}\,(1-\lambda_l)}{1-\lambda_l} = \hat{c} \end{aligned}$$

This finally delivers the contradiction with the assumption that the linear combination from the beginning of the proof had minimal cost.

Part 2:

The second part of the proof shows $\forall i : S_i \cap V^{\prec}(S) = \emptyset$. Using the result from the first part, $\forall i : S_i \subseteq V^{\preceq}(S)$, we can rewrite Equation (B.3.4) *restricted to nodes* $v \in S$ as:

$$\sum_{\forall i : v \in \mathbb{P}_+^{-1}(S_i) \cap S} \lambda_i = \sum_{\forall i : v \in S_i \cap S} \lambda_i \geq 1 \qquad \forall v \in S \subseteq \mathbb{P}_+^{-1}(S)$$

If we multiply this inequality by $c(v)$ and sum over all nodes of a subset $S' \subseteq S$ we obtain:

$$\sum_{v\in S'} c(v) \sum_{\forall i: v\in S_i \cap S} \lambda_i \geq c(S') \tag{B.3.6}$$

This inequality holds for all subsets of S and is used in the following derivation to prove $\forall i : S_i \subseteq S$. It starts with Equ. (B.3.5):

$$\begin{aligned} c(S) &\geq \sum_{i=1}^{k} \lambda_i c(S_i) \\ &= \sum_{i=1}^{k} \lambda_i \left(c(S_i \cap S) + c(S_i \setminus S)\right) \\ &= \underbrace{\sum_{v\in S} c(v) \sum_{\forall i: v\in S_i \cap S} \lambda_i}_{\geq c(S) \text{ with (B.3.6)}} + \underbrace{\sum_{i=1}^{k} \lambda_i c(S_i \setminus S)}_{=0} \end{aligned}$$

The last sum must be zero, and can only be zero if $\forall i : S_i \subseteq S$.

Part 3:

We have shown in this proof so far that all node cuts $S_1, \ldots, S_k$ must be subsets of S. Let S_z be an arbitrary one of these node cuts. Because S is atomic, there exists a $w \in \mathbb{P}_+^{-1}(S)$ such that $w \notin \mathbb{P}_+^{-1}(S_z)$ and $w \notin \mathbb{P}_+^{-1}(S \setminus S_z)$, and with this:

$$S_z \cap V^{\succeq}(w) \neq \emptyset \wedge (S \setminus S_z) \cap V^{\succeq}(w) \neq \emptyset \tag{B.3.7}$$

Let $J := \left\{ i \,\middle|\, w \in \mathbb{P}_+^{-1}(S_i) \right\}$, then $z \notin J$ and Equation (B.3.4) for the node w can be written as:

$$\sum_{i\in J} \lambda_i \geq 1 \tag{B.3.8}$$

We use this inequality to derive a contradiction and finally conclude the proof with this. Again, we start from Equ. (B.3.5):

$$\begin{aligned} c(S) &\geq \sum_{i=1}^{k} \lambda_i c(S_i) = \sum_{v\in S} c(v) \sum_{\forall i: v\in S_i} \lambda_i \\ &= \sum_{v\in S\cap V^{\succeq}(w)} c(v) \sum_{\forall i: v\in S_i} \lambda_i + \sum_{v\in S\setminus V^{\succeq}(w)} c(v) \sum_{\forall i: v\in S_i} \lambda_i \\ &= \underbrace{\sum_{v\in S\cap V^{\succeq}(w)} c(v) \sum_{\substack{\forall i: v\in S_i \\ i\in J}} \lambda_i}_{\geq c(S\cap V^{\succeq}(w))} + \underbrace{\sum_{v\in S\cap V^{\succeq}(w)} c(v) \sum_{\substack{\forall i: v\in S_i \\ i\notin J}} \lambda_i}_{=0} + \underbrace{\sum_{v\in S\setminus V^{\succeq}(w)} c(v) \sum_{\forall i: v\in S_i} \lambda_i}_{\geq c(S\setminus V^{\succeq}(w))} \end{aligned}$$

The first double sum in the last row is greater or equal to $c(S \cap V^{\succeq}(w))$ because it follows from (B.3.8) that for each $v \in S \cap V^{\succeq}(w)$ the interior sum is greater or equal to one—the additional constraint $i : v \in S_i$ compared to (B.3.8) causes no restriction since $i \in J \Rightarrow S_i \supseteq S \cap V^{\succeq}(w)$. The estimate for the last double sum is taken from (B.3.6).

Then the middle double sum must be empty since $c(S) = c(S \cap V^{\succeq}(w)) + c(S \setminus V^{\succeq}(w))$. However, (B.3.7) states that there is at least one index such that $S_i \cap V^{\succeq}(w) \neq \emptyset$ and $i \notin J$: z ■

Bibliography

[ACG$^+$04] Lelac Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding Effective Compilation Sequences. In *Proceedings of the ACM SIGPLAN 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 231–239, Washington, DC, USA, June 2004.

[Adv04] Advanced Micro Devices, Inc. *Software Optimization Guide for AMD Athlon^{TM} 64 and AMD Opteron^{TM} Processors*, March 2004.

[AG01] Andrew W. Appel and Lal George. Optimal Spilling for CISC Machines with Few Registers. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, pages 243–253, Snowbird, Utah, USA, June 2001.

[Alp03] Don Alpert. Itanium Processor Status Report. *In-Stat/MDR Microprocessor Report*, July 2003.

[ANS89] ANSI, New York. *American National Standard Programming Language C, ANSI X3.159-1989*, December 14 1989.

[ASMC98] Pritpal S. Ahuja, Kevin Skadron, Margaret Martonosi, and Douglas W. Clark. Multipath Execution: Opportunities and Limits. In *Proceedings of the 12th International Conference on Supercomputing*, pages 101–108, Melbourne, Australia, 1998.

[AZ96] Nina Amenta and Günter Ziegler. Shadows and Slices of Polytopes. In *Proceedings of the 12th Annual ACM Symposium on Computational Geometry*, pages 10–19, Philadelphia, Pennsylvania, 1996.

[Bas95] S. Bashford. Code Generation Techniques for Irregular Architectures. Technical Report 596, University of Dortmund, 1995.

[BCC$^+$00] Jay Bharadwaj, William Y. Chen, Weihaw Chuang, Gerolf Hoflehner, Kishore Menezes, Kalyan Muthukumar, and Jim Pierce. The Intel IA-64 Compiler Code Generator. *IEEE Micro*, 20(5):44–53, 2000.

[BCGS96] David A. Berson, Pohua P. Chang, Rajiv Gupta, and Mary Lou Soffa. Integrating Program Optimizations and Transformations with the Scheduling of Instruction Level Parallelism. In *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 207–221, San Jose, USA, August 1996. Springer-Verlag.

[Bea91] S.J. Beaty. *Instruction Scheduling Using Genetic Algorithms*. PhD thesis, Department of Mechanical Engineering, Colorado State University, Fort Collins, USA, 1991.

[BFG$^+$00] Robert E. Bixby, Mary Fenelon, Zonghao Gu, Ed Rothberg, and Roland Wunderling. ILOG CPLEX Division. MIP: Theory and Practice - Closing the Gap. In M. J. D. Powell and S. Scholtes, editors, *System Modelling and Optimization: Methods, Theory and Applications*, pages 19–49. Kluwer, The Netherlands, 2000.

[BGS98] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Integrated Instruction Scheduling and Register Allocation Techniques. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 247–262, Chapel Hill, NC, USA, August 1998. Springer-Verlag.

[BL99a] Steven Bashford and Rainer Leupers. Constraint Driven Code Selection for Fixed-Point DSPs. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation*, pages 817–822, New Orleans, Louisiana, USA, 1999.

[BL99b] Steven Bashford and Rainer Leupers. Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths. *Design Automation for Embedded Systems (Kluwer Academic Publishers)*, 4(2-3):119–165, 1999.

[Bli96] T. Blickle. *Theory of Evolutionary Algorithms and Applications to System Design*. PhD thesis, ETH Zürich, 1996.

[BMA02] Ivan D. Baev, Waleed M. Meleis, and Santosh G. Abraham. Backtracking-Based Instruction Scheduling to Fill Branch Delay Slots. *International Journal of Parallel Programming (Kluwer Academic Publishers)*, 30(6):397–418, 2002.

[BMM00] J. Bharadwaj, K. Menezes, and C. McKinsey. Wavefront Scheduling: Path Based Data Representation and Scheduling of Subgraphs. *Journal of Instruction-Level Parallelism*, 1(6):1–6, 2000.

[BMS02] David Bradley, Patrick Mahoney, and Blaine Stackhouse. The 16kb Single-Cycle Read Access Cache on a Next Generation 64b Itanium^{TM} Microprocessor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, San Francisco, February 2002.

[BR91] David Bernstein and Michael Rodeh. Global Instruction Scheduling for Superscalar Machines. *Proceedings of the ACM SIGPLAN '91 on Programming Language Design and Implementation (PLDI)*, June 1991.

[BRS92] D. Bernstein, M. Rodeh, and M. Sagiv. Proving Safety of Speculative Load Instructions at Compile-Time. In *Proceedings of the 4th European Symposium on Programming (ESOP)*, Rennes, France, February 1992.

[CCK97] C-M. Chang, C-M. Chen, and C-T. King. Using Integer Linear Programming for Instruction Scheduling and Register Allocation in Multi-Issue Processors. *Computers and Mathematics with Applications*, 34(9):1–14, November 1997.

[CCL$^+$96] Pohua P. Chang, Dong-Yuan Chen, Yong-Fong Lee, Youfeng Wu, and Utpal Banerjee. Bidirectional Scheduling: A New Global Code Scheduling Approach. In *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 222–230, San Jose, USA, August 1996. Springer-Verlag.

[CHN99] Marius Cornea-Hasegan and Bob Norin. IA-64 Floating-Point Operations and the IEEE Standard for Binary Floating-Point Arithmetic. *Intel Technology Journal*, Q4, 1999.

[CKGN01] Youngsoo Choi, Allan Knies, Luke Gerke, and Tin-Fook Ngai. The Impact of If-conversion and Branch Prediction on Program Execution on the Intel® Itanium® Processor. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 182–191, Austin, Texas, December 2001.

[CL03] Jean-Francois Collard and Daniel Lavery. Optimizations to Prevent Cache Penalties for the Intel® Itanium® 2 Processor. In *Proceedings of the First IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, San Francisco, March 2003.

[CLF$^+$03] Dong-Yuan Chen, Lixia Liu, Chen Fu, Shuxin Yang, Chengyong Wu, and Roy Ju. Efficient Resource Management during Instruction Scheduling for the EPIC Architecture. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, New Orleans, September 2003.

[CLJ$^+$04] Dong-Yuan Chen, Lixia Liu, Roy D.C. Ju, Chen Fu, Shuxin Yang, and Chengyong Wu. Efficient Modeling of Itanium® Architecture during Instruction Scheduling using Extended Finite State Automata. *Journal of Instruction-Level Parallelism*, 2004. www.jilp.org.

[CLL$^+$02] Gianpiero Cabodi, Mihai Lazarescu, Luciano Lavagno, Sergio Nocco, Claudio Passerone, and Stefano Quer. A Symbolic Approach for the Combined Solution of Scheduling and Allocation. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS)*, pages 237–242, Kyoto, Japan, October 2002.

[CLR01] Thomas H. Cormen, Charles E. Leierson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts;London, England, 2001.

[CS98] Keith D. Cooper and Philip J. Schielke. Non-Local Instruction Scheduling with Limited Code Growth. In *Proceedings of the 1998 ACM Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 193–207, Montreal, Canada, June 1998.

[CSS98] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. An Experimental Evaluation of List Scheduling. Technical Report TR98-326, Rice University, September 1998.

[CSS99] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for Reduced Code Space Using Genetic Algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, Atlanta, USA, May 1999.

[CWM93] S. Chaudhuri, R.A. Walker, and J.E. Mitchell. The Structure of Assignment, Precedence, and Resource Constraints in the ILP Approach to the Scheduling Problem. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pages 25–31, Cambridge, USA, October 1993.

[CWM94] S. Chaudhuri, R.A. Walker, and J.E. Mitchell. Analyzing and Exploiting the Structure of the Constraints in the ILP-Approach to the Scheduling Problem. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):456–471, December 1994.

[Dak65] R.J. Dakin. A Tree-Search Algorithm for Mixed Integer Programming Problems. *The Computer Journal*, 8:250–255, 1965.

[Dan51] G.B. Dantzig. Maximization of a Linear Function of Variables Subject to Linear Inequalities. In Tj. C. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 339–347. Wiley, New York, 1951.

[DKK+99] Carole Dulong, Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, John Ng, and David Sehr. An Overview of the Intel® IA-64 Compiler. *Intel Technology Journal*, (Q4), 1999.

[Eis00] F. Eisenbrand. *Gomory-Chvatal Cutting Planes and the Elementary Closure of Polyhedra*. PhD thesis, Saarland University, 2000.

[EM92] K. Ebcioglu and S. Moon. An Efficient Resource Constrained Global Scheduling Technique for Superscalar and VLIW Processors. In *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Portland, Oregon, USA, November 1992.

[Epp95] David Eppstein. Subgraph Isomorphism in Planar Graphs and Related Problems. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 632–640, San Francisco, 1995. Society for Industrial and Applied Mathematics.

[Fis81] J.A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[FO02] Eric S. Fetzer and John T. Orton. A Fully-Bypassed 6-Issue Integer Datapath and Register File on an Itanium^{TM} Microprocessor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, San Francisco, February 2002.

[FW02] Changqing Fu and Kent Wilken. A Faster Optimal Register Allocator. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 245–256, Istanbul, Turkey, November 2002.

[GB03] Lal George and Matthias Blume. Taming the IXP Network Processor. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 26–37, San Diego, USA, June 2003.

[GE92] C.H. Gebotys and M.I. Elmasry. *Optimal VLSI Architectural Synthesis: Area, Performance and Testability*. Kluwer Academic, 1992.

[GE93] C.H. Gebotys and M.I. Elmasry. Global Optimization Approach for Architectural Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1266–1278, 1993.

[Geb97] Catherine H. Gebotys. An Efficient Model for DSP Code Generation: Performance, Code Size, Estimated Energy. In *Proceedings of the 10th International Symposium on System Synthesis (ISSS)*, pages 41–47, Antwerp, Belgium, September 1997.

[GH88] J. Goodman and W. Hsu. Code Scheduling and Register Allocation. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, USA, June 1988.

[GJ79] M. Garey and D.S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. Freemann and Company, 1979.

[GLS01] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the Importance of Points-to Analysis and Other Memory Disambiguation Methods for C Programs. *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI)*, pages 47–58, June 2001.

[GNU] GNU Project – Free Software Foundation. www.gnu.org.

[Gom58] R.E. Gomory. Outline of an Algorithm for Integer Solutions to Linear Programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958.

[GS90] R. Gupta and M.L. Soffa. Region Scheduling: An Approach for Detecting and Redistributing Parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, 1990.

[Gup98] Rajiv Gupta. A Code Motion Framework for Global Instruction Scheduling. In *Proceedings of the 7th International Conference on Compiler Construction (CC)*, pages 219–233, Lisbon, Portugal, March 1998. Springer-Verlag.

[Hag97] Torben Hagerup. Datenstrukturen und Algorithmen, Forschungsbericht. Technical report, Max-Planck-Institut für Informatik, July 1997. MPI-I-97-1-013, In German.

[HB01] Steve Haga and Rajeev Barua. EPIC Instruction Scheduling Based on Optimal Approaches. In *Proceedings of the EPIC-1 Workshop*, Austin, Texas, December 2001.

[HCLJ01] Yuan-Shin Hwang, Peng-Sheng Chen, Jenq Kuen Lee, and Roy Dz-Ching Ju. Probabilistic Points-to Analysis. In *Lecture Notes in Computer Science: Proc. of the 14th Workshop on Languages and Compilers for Parallel Computing (LCPC'2001)*, volume 2624, pages 290–305, Kumberland Falls, KY, August 2001. Springer.

[HKST99] J. Harrison, T. Kubaska, S. Story, and P.T.P. Tang. The Computation of Transcendental Functions on the IA-64 Architecture. *Intel Technology Journal*, Q4, 1999.

[HLH91] Cheng-Tsung Hwang, Jiahn-Hurng Lee, and Yu-Chin Hsu. A Formal Approach to the Scheduling Poblem in High Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(4):464–475, 1991.

[HLW00] M. Heffernan, J. Liu, and K. Wilken. Optimal Instruction Scheduling Using Integer Programming. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 121–133, Vancouver, Canada, June 2000.

[HMC$^+$93] W.-m. W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, and D.M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing (Kluwer Academic Publishers)*, pages 229–248, 1993.

[Hop00] Martin Hopkins. Guest Viewpoint: A Critical Look at IA-64. *In-Stat/MDR Microprocessor Report*, February 2000.

[HP03] J.L. Hennessy and D.A. Patterson. *Computer Architecture: a Quantitive Approach*. Morgan Kaufmann, San Francisco, third edition, 2003.

[HSU$^+$01] Glenn Hinton, Dave Sager, Mike Upton, Darrel Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The Microarchitecture of the Pentium® 4 Processor. *Intel Technology Journal*, (Q1), 2001.

[ILO03a] ILOG S. A., Paris. *ILOG Concert Technology 2.0 Reference Manual*, October 2003.

[ILO03b] ILOG S. A., Paris. *ILOG CPLEX 9.0 User's Manual*, October 2003.

[Int01a] Intel. *IA-64 Software Conventions and Runtime Architecture Guide*, May 2001.

[Int01b] Intel. *Intel® Itanium^{TM} Architecture Assembly Language Reference Guide*, 2001.

[Int02a] Intel. *Intel® Itanium® Architecture Software Developer's Manual, Volume 1: Application Architecture*, October 2002.

[Int02b] Intel. *Intel® Itanium® Architecture Software Developer's Manual, Volume 2: System Architecture*, October 2002.

[Int02c] Intel. *Intel® Itanium® Architecture Software Developer's Manual, Volume 3: Instruction Set Reference*, October 2002.

[Int03] Intel. *Intel® C++ Compiler for Linux Systems User's Guide*, 2003. Compiler version 8.0.

[Int04] Intel. *Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization*, May 2004.

[Jai91] Suneel Jain. Circular Scheduling: a new Technique to Perform Software Pipelining. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–228, Toronto, Canada, 1991.

[JCO98] R. D.-C. Ju, J.-F. Collard, and K. Oukbir. Probabilistic Memory Disambiguation and its Application to Data Speculation. In *Proc. of the 3rd Workshop on Interaction between Compilers and Computer Architectures*, San Jose, October 1998.

[JH00] William S. Worley Jr. and Jerry Huck. Guest Viewpoint: Is Out-of-order Out of Date? *In-Stat/MDR Microprocessor Report*, February 2000.

[Kar84] N. Karmarkar. A new Polynomial-Time Algorithm for Linear Programming. *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 302–311, 1984.

[Käs00a] Daniel Kästner. *Retargetable Code Optimisation by Integer Linear Programming*. PhD thesis, Saarland University, 2000.

[Käs00b] Daniel Kästner. PROPAN: A Retargetable System for Postpass Optimisations and Analyses. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES)*, Vancouver, Canada, June 2000.

[KB01] Christoph Keßler and Andrzej Bednarski. A Dynamic Programming Approach to Optimal Integrated Code Generation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 165–174, Snowbird, Utah, USA, June 2001.

[KE93] Daniel R. Kerns and Susan J. Eggers. Balanced Scheduling: Instruction Scheduling When Memory Latency is Uncertain. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI)*, pages 278–289, Albuquerque, NM, USA, June 1993.

[Kha80] L.G. Khachiyan. Polynomial Algorithms in Linear Programming (in Russian). *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 20:51–68, 1980.

[KW01] Daniel Kästner and Sebastian Winkel. ILP-based Instruction Scheduling for IA-64. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Snowbird, June 2001.

[Lav97] Daniel M. Lavery. *Modulo Scheduling for Control-Intensive General-Purpose Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[LC02] Jack Liu and Fred Chow. A Near-Optimal Instruction Scheduler for a Tightly Constrained, Variable Instruction Set Embedded Processor. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 9–18, Grenoble, France, October 2002.

[LCDT96] Raymond Lo, Sun Chan, James C. Dehnert, and Ross A. Towle. Aggregate Operation Movement: A Min-Cut Approach to Global Code Motion. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing*, volume II, pages 801–814, Lyon, France, August 1996. Springer-Verlag.

[LDMM02] Terry Lyon, Eric Delano, Cameron McNairy, and Dean Mulla. Data Cache Design Considerations for the Itanium® 2 Processor. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*, Freiburg, Germany, September 2002.

[Leu00] Rainer Leupers. Code Generation for Embedded Processors. In *Proceedings of the 13th International Symposium on System Synthesis (ISSS)*, pages 173–178, Madrid, Spain, September 2000. IEEE Computer Society.

[Li01] Wei Li. Compiling for Itanium Architecture: Triumphs and Challenges. http://systems.cs.colorado.edu/EPIC1, December 2001. EPIC-1 keynote slides.

[LM97] Rainer Leupers and Peter Marwedel. Time-Constrained Code Compaction for DSPs. *IEEE Transactions on VLSI Systems*, 5(1), September 1997.

[LM04] Markus Lorenz and Peter Marwedel. Phase Coupled Code Generation for DSPs Using a Genetic Algorithm. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'04)*, volume II, Paris, France, February 2004. IEEE Computer Society.

[LMD94] Birger Landwehr, Peter Marwedel, and Rainer Dömer. OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming. In *Proceedings of the EURODAC '94*, pages 90–95, Grenoble, France, September 1994.

[LWT00] Hsien-Hsin Lee, Youfeng Wu, and Gary Tyson. Quantifying Instruction-Level Parallelism Limits on an EPIC Architecture. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 21–27, Austin, Texas, April 2000.

[MCWL01] Kalyan Muthukumar, Dong-Yuan Chen, Youfeng Wu, and Daniel M. Lavery. Software Pipelining of Loops with Early Exits for IA-64. In *Proceedings of the 1st Workshop on EPIC Architectures and Compiler Technology (EPIC-1)*, Austin, Texas, December 2001.

[MF02] Seda Ogrenci Memik and Farzan Fallah. Accelerated SAT-Based Scheduling of Control/Data Flow Graphs. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*, Freiburg, Germany, September 2002.

[MG95] P. Marwedel and G. Goossens. *Code Generation for Embedded Processors*. Kluwer, Boston; London; Dortrecht, 1995.

[MK02] James McCormick and Allan Knies. A Brief Analysis of the SPEC CPU2000 Benchmarks on the Intel® Itanium® 2 Processor. *HotChips 14*, 2002.

[ML02] John Markoff and Steve Lohr. Intel's Huge Bet Turns Iffy. *The New York Times*, September 29, 2002.

[MLC$^+$92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO)*, pages 45–54, Portland, Oregon, USA, December 1992.

[MP00] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture. Complexity and Correctness*. Springer, Berlin;Heidelberg;New York, 2000.

[MS03] Cameron McNairy and Don Soltis. Itanium 2 Processor Microarchitecture. *IEEE Micro*, March 2003.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, Kalifornien, 1997.

[NH02] Samuel D. Naffziger and Gary Hammond. The Implementation of the Next Generation 64b Itanium^{TM} Microprocessor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, San Francisco, February 2002.

[Nic85] A. Nicolau. Uniform Parallelism Exploitation in Ordinary Programs. In *International Conference on Parallel Processing*, pages 614–618. IEEE Computer Society Press, August 1985.

[NN94] S. Novack and A. Nicolau. Mutation scheduling: A Unified Approach to Compiling for Fine-Grain Parallelism. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 16–30. Springer LNCS, 1994.

[NP03] V. Krishna Nandivada and Jens Palsberg. Efficient Spill Code for SDRAM. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 24–31, San Jose, USA, October 2003.

[NP04] Mayur Naik and Jens Palsberg. Compiling with Code-Size Constraints. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(1):163–181, 2004.

[NR01] M. Narasimhan and J. Ramanujam. A Fast Approach to Computing Exact Solutions to the Resource-Constrained Scheduling Problem. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(4):490–500, 2001.

[NW88] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.

[NW89] G.L. Nemhauser and L.A. Wolsey. Integer Programming. In G.L. Nemhauser, A.H.G. R. Kan, and M.J. Todd, editors, *Handbooks in Operations Research and Management Science*, chapter VI, pages 447–527. North-Holland, Amsterdam; New York; Oxford, 1989.

[Pad73] Manfred W. Padberg. On the Facial Structure of Set Packing Polyhedra. *Mathematical Programming*, 5:199–215, 1973.

[PGTM99] Matthew A. Postiff, David A. Greene, Gary S. Tyson, and Trevor N. Mudge. The Limits of Instruction Level Parallelism in SPEC95 Applications. *SIGARCH Computer Architecture News*, 27(1):31–34, March 1999.

[RDGG04] Hongbo Rong, Alban Douillet, R. Govindarajan, and Guang R. Gao. Code generation for single-dimension software pipelining of multi-dimensional loops. In *Proceedings of the Second IEEE/ACM International Symposium on Code Generation and Optimization*, Palo Alto, California, 2004. IEEE Computer Society.

[RF93] B.R. Rau and J.A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[RG02] Reid Riedlinger and Tom Grutkowski. The High Bandwidth, 256KB 2nd Level Cache on an Itanium^{TM} Microprocessor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, San Francisco, February 2002.

[RGSL96] J. Ruttenberg, G.R. Gao, A. Stoutchinin, and W. Lichtenstein. Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Languages Design and Implementation (PLDI)*, pages 1–11, Philadelphia, USA, May 1996.

[RML$^+$01] Ronny Ronen, Avi Mendelson, Konrad Lai, Shih-Lien Lu, Fred Pollack, and John P. Shen. Coming Challenges in Microarchitecture and Architecture. *Proceedings of the IEEE*, 89(3), March 2001.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, Chichester; New York; Brisbane, 1986.

[Sch03a] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency; Matroids, Trees, Stable Sets*, volume B. Springer, Berlin; Heidelberg; New York, 2003.

[Sch03b] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency; Paths, Flows, Matchings*, volume A. Springer, Berlin; Heidelberg; New York, 2003.

[SL96] Mark G. Stoodley and Corinna G. Lee. Software Pipelining Loops with Conditional Branches. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 262–273, Paris, France, 1996. IEEE Computer Society.

[SPE00] SPEC CPU 2000 Benchmark. www.spec.org, 2000.

[SR03] Jason Stinson and Stefan Rusu. A 1.5GHz Third Generation Itanium® 2 Processor. In *Proceedings of the 40th Design Automation Conference*, Anaheim, USA, June 2003.

[SRM$^+$94] Michael S. Schlansker, B. Ramakrishna Rau, Scott Mahlke, Vinod Kathail, Richard Johnson, Sadun Anik, and Santosh G. Abraham. Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity. Technical Report HPL-96-120, HP Labs, November 1994.

[SS02] Y. N. Srikant and Priti Shankar, editors. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, Boca Raton, 2002.

[Ste03] Ingmar Stein. Bundling for IA-64. Technical report, Saarland University, August 2003. FoPra-Bericht. In German.

[SW04] Peter Sanders and Sebastian Winkel. Super Scalar Sample Sort. In *Proceedings of the 12th European Symposium on Algorithms (ESA)*, Bergen, Norway, 2004. Springer LNCS.

[TDF$^+$02] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1), 2002.

[Tri00] Walter A. Triebel. *Itanium Architecture for Software Developers*. Intel Press, July 2000.

[vBW01] Peter van Beek and Kent D. Wilken. Fast Optimal Instruction Scheduling for Single-Issue Processors with Arbitrary Latencies. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 625–639, Paphos, Cyprus, 2001. Springer-Verlag.

[WGB94] T.C. Wilson, G.W. Grewal, and D.K. Banerji. An ILP Solution for Simultaneous Scheduling, Allocation, and Binding in Multiple Block Synthesis. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pages 581–586, Cambridge, USA, October 1994. IEEE Computer Society Press.

[WGHB95] T. Wilson, G. Grewal, S. Henshall, and D. Banerji. An ILP-Based Approach to Code Generation. In *[MG95]*, chapter 6, pages 103–118. 1995.

[Win01] Sebastian Winkel. ILP-basierte Instruktionsanordnung für IA-64. Master's thesis, Saarland University, 2001. In German.

[Win02] Sebastian Winkel. Optimal Global Scheduling for Itanium Processor Family. In *Proceedings of the EPIC-2 Workshop*, Istanbul, November 2002.

[Win04] Sebastian Winkel. Exploring the Performance Potential of Itanium® Processors with ILP-based Scheduling. In *Proceedings of the Second IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Palo Alto, March 2004.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[WM97] R. Wilhelm and D. Maurer. *Übersetzerbau. Theorie, Konstruktion, Generierung; zweite, überarbeitete und erweiterte Auflage*. Springer, Berlin; Heidelberg; New York, 1997. In German.

[YP91] T. Y. Yeh and Y. N. Patt. Two-Level Adaptive Training Branch Prediction. In *Proceedings of the 24th International Symposium on Microarchitecture (MICRO)*, pages 51–61, New York, November 1991.

[ZCS03] Min Zhao, Bruce Childers, and Mary Lou Soffa. Predicting the Impact of Optimizations for Embedded Systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–11, San Diego, USA, June 2003.

[Zha96] L. Zhang. *SILP. Scheduling and Allocating with Integer Linear Programming*. PhD thesis, Saarland University, 1996.

[ZJC01] Huiyang Zhou, Matthew D. Jennings, and Thomas M. Conte. Tree Traversal Scheduling: A Global Instruction Scheduling Technique for VLIW/EPIC Processors. In *Proceedings of the 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 223–238, Cumberland Falls, KY, USA, August 2001. Springer-Verlag.

[ZTB00] Eckart Zitzler, Jürgen Teich, and Shuvra S. Bhattacharyya. Evolutionary Algorithms for the Synthesis of Embedded Software. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(4):452–456, August 2000.

Appendix C

List of Symbols

Alphabetic

Nonalphabetic

ILP Variables

Index