Generation of Distributed Supervisors for Parallel Compilers

Martin Alt[‡]
Georg Sander[‡]
Reinhard Wilhelm

Technischer Bericht **A01/93**
Universität des Saarlandes
FB 14 Informatik
6600 Saarbrücken

February 17, 1993

Abstract

This paper presents a new approach towards solving the combination and communication problems between different compiler tasks. As optimizations may generate as well as destroy application conditions of other tasks a carefully chosen application order is important for the effectiveness of the compiler system. Each task is solved by exactly one implementation (engine) and is characterized by its input-output behaviour and an optional heuristics. The specification of all tasks in this manner allows the generation of distributed supervisors for the whole compilation system. The result is a clear semantics of the compiler behaviour during compilation and the separation of algorithm and communication. Software engineering advantages are the easy integration of independently developed parts and the reusability of code. The flexibility of such a compiler system results in high portability even across hardware architectures and topologies.

[‡]Funded by the **ESPRIT** Project #5399 (COMPARE)

Contents

1 Introduction

The ESPRIT project #5399 COMPARE (Compiler Generation for Parallel Architectures) aims at optimizing compilers for different source–target combinations with special emphasis on instruction level parallel targets. The currently chosen source languages are C, Fortran, ML and a parallel version of Modula II, Modula-P. As targets are envisioned the MIPS 4000, the 68040, the SuperScalar SPARC (Viking) and the T9000 processor. A new compiler organization (**CoSy**) has been designed and implemented. It is based on a modularization of compilers into engines(see [17]), which each solve a certain subtask of compilation. Engines have explicitly specified input and output data and no side effects. This organization allows to easily configure a compiler out of existing or newly written/generated engines and to integrate new engines into an existing compiler. In addition, it provides for portability across host topologies. We believe, that this approach to the reuse of software is not confined to compilers, but that it can be used in other software architectures.

A collection of engines, in particular middle end engines in optimizing compilers, may have a quite complex or even cyclic dependence relation. This represents the experience of compiler writers, that some optimization enables another optimization, which in turn enables another or an earlier used optimization.

Semantically, the optimizer loop is represented by cyclic dependences between engines. The computation of this loop needs a way of detecting termination, in our case implied by convergence of the states of the compiler's data structures. Our approach is generative; a functional specification of the input-output behaviour on the compiler's data structures is given for all engines. Taken together, the engines' specification form a recursive system corresponding to the above loop.

The implementation, however, is not functional but imperative; the compiler works on its data structures destructively. This necessitates a synchronization scheme between engines, which guarantees the correctness of the implementation with respect to the functional specification. In our case, the correct cooperation of the engines is implied by requiring the consistency of the data structures between engine activations. The space of possible engine cooperation schemes satisfying the consistency requirements is still very big. Consecutive activations of the same engine without effect on the data structures would be a correct, but undesirable case. In a next step, compilation speed is increased by chosing scheduling strategies minimizing engine activations and avoiding redundant computations. Specified heuristics are used for that purpose.

The activation of engines and their cooperation on the compiler's data structures is controlled by what is called *supervision.*

An engine description compiler (EDC) generates a supervisor envelope around each engine code from its specification, which guarantees the coordinated access to the data structures. This is called *distributed supervision.*

The supervisor may include tests for convergence and for termination in the case of cyclic dependences.

Different supervisors can be generated for different host topologies. Currently, uniprocessor and shared memory multiprocessor hosts are supported. On the latter, only engine level, coarse grain parallelism is implemented. Data parallel and pipelined compiler organizations are subject of ongoing research.

2 Dependences between Compiler Engines

A CoSy compiler consists of a collection of engines together with a set of data structures. Engines may be activated in many different orders, sequentially or in parallel. Not all such orders make sense. *Reasonable activation orders* can be specified by a dependence relation on the set of engines and the set of data structures. This relation defines a bipartite graph. In general, an engine can have three different access methods on a data structure: *produce*, *use* and *modify*. *use* dependences always go from data structures to engines and *modify*, *produce* edges from engines to data structures.

Modify dependences are critical for the scheduling of compiler engine activations; when an activation of engine x has modified data structure y, other engines (transitively) depending on y may have to be rescheduled. Thus, *modify* dependences cause the well known *phase ordering problem* in compiler construction (also called optimizer loop). A clever rescheduling strategy reduces the necessary amount of recomputations to (nearly) a minimum.

We now formally define the dependence graph. This graph is the starting point for the above mentioned analyses, the optimization of recomputations, and the generation of supervision code.

Definition: 1 (engine dependence graph)
An engine dependence graph is a bipartite graph EDG = (V,E). V consists of the set of engines and the set of data structures accessed by these engines. E consists of the following (typed) edges:

- $e \overset{\text{modify}}{\rightarrow} d$, *if engine e modifies data structure d*
- $e \overset{\text{produce}}{\rightarrow} d$, *if data structure d is produced by engine e*
- $d \overset{\text{use}}{\rightarrow} e$, *if data structure d is used by engine e*

The *engine dependence graph* summarizes all information necessary for the generation of correct synchronization. Not all possible EDG's can be given a semantics, but there is a useful subset of such graphs. A *well defined* EDG is an EDG with the following properties:

1. any data structure has exactly one incoming *produce* edge; that means each data structure is produced by exactly one engine. Section 10 discusses a possible removal of this restriction.

2. an engine e producing data structure d is not specified as using or modifying d.

3. there exist no *produce–use* cycles.

4. An engine e, modifying a data structure d, is also specified as using d.

We will now give an example demonstrating a simple compiler (without code generation). We assume nine engines with the following input-output behaviour.

Example: 1 The functionality of the compiler engines is as follows:

- *Scan*: It implements the lexical analysis, i.e. the transformation of the Character Stream of a source program into a Token Stream using a finite state automaton. It is generated from a specification of regular expressions (see [1],[17]).

- *Parse*: It implements the syntactical analysis of a source program and produces the Syntax Tree from the Token Stream. It is generated from a parser specification (see [1], [17]).

- *ControlFlow*: It constructs the (interprocedural) Control Flow Graph from the Syntax Tree. An introduction to the theory of control flow graphs can be found in [6] and [2].

- *DefUse*: It produces the Def-Use sets from the Syntax Tree.

- *Live*: This engine implements dead variable elimination (see [6]). It modifies the Syntax Tree using the Control Flow Graph.

- *ModVars*: computes the (upper approximative) set of variables that may be modified by a procedure call. It propagates the Def-Use sets over the call graph (local data structure of that engine). It uses the Syntax Tree, Def-Use Sets and produces the Mod-Var sets.

- *ConstProp*: computes the sets of constant variables, Constant sets, by using the Control Flow Graph, the Def-Use sets and the Mod-Var sets (i.e. interprocedural). It implements a slightly modified algorithm of [3].

- *ConstFold*: The engine modifies the Syntax Tree replacing expressions known to have constant values by their value, which is computed by an integrated interpreter of the language. It uses the Constant sets and the Control Flow Graph.

- *TreeSimpl* modifies the Syntax Tree according to some specified (syntactical) transformation rules.

The *engine dependence graph* of that scenario with the formerly mentioned engines and data structures `Character Stream`, `Token Stream`, `Syntax Tree`, `Control Flow Graph`, `Constant sets`, `Mod-Var sets` and `Def-Use sets` is given in Figure 1. The solid boxes denote data structures. The engines are drawn using ellipses.

After specifying the access methods for each engine, we can deduce a useful kind of dependence between engines. We say, an engine e_2 is *true* dependent on another engine e_1, if there is a *produce* edge from e_1 to a data structure d and a *use* edge from d to e_2. The engines an engine e is *true* dependent on are defined by:

$$\mathrm{PU}(e) = \{\ e' \mid \exists \text{ an edge } e' \overset{\text{produce}}{\rightarrow} d \text{ and } d \overset{\text{use}}{\rightarrow} e\ \}$$

The *true* dependences define a useful partial order on the engines; the *phase ordering problem* for this special compiler consists in finding a sequence of engine activations that does not violate this partial order. This sequence is in close relationship with at least one possible compiler.

The partial order limits the possible amount of (coarse grain) parallelism in a compiler. It is in sequential compilers artificially completed to a total one by the ordering of engine calls.

3 The Abstract Framework

We will now give the theoretical background for the phase ordering problem. Formally the optimal translation from a source to a target language can be defined as a minimization problem.

Let $e_1, \ldots, e_a$ be the set of engines available in a compiler system. Let $D_1, \ldots, D_l$ be the types of the used data structures (mainly the intermediate representations together with data flow sets). Two types are considered as special namely D_i, the input and D_o, the output type of the compiler system. Traditionally, D_i is the source language (to be translated) and D_o is the machine language of the target computer. Each engine e_j of the compiler performs a function:

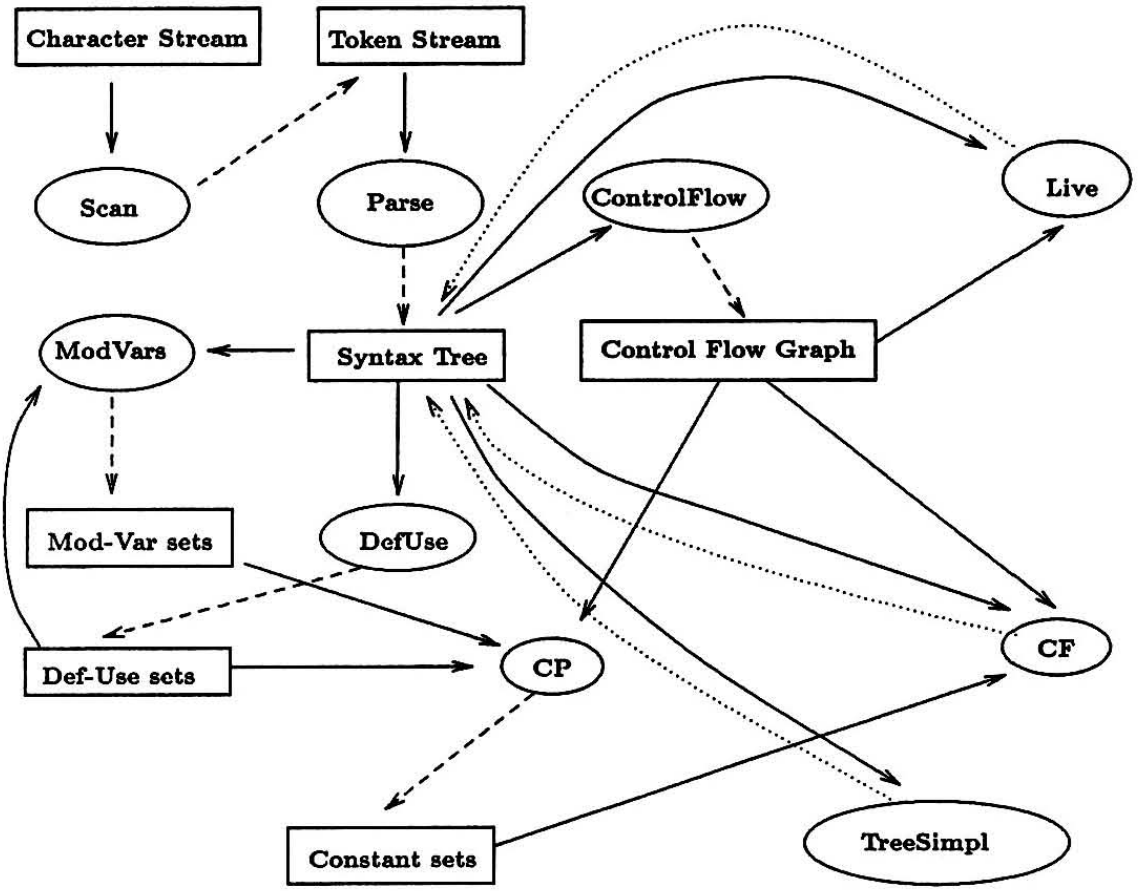
$$f_{e_j} :\ D_1 \times \ldots \times D_l \rightarrow D_1 \times \ldots \times D_l \quad , with$$

$$f_{e_j}(v_1^1, \ldots, v_l^1) = (v_1^2, \ldots, v_l^2) \quad , v_x^i \in D_x, i \in \{1,2\}$$

To determine the *quality* of the produced data structures, there must exist cost functions $g_x : D_x \rightarrow \mathcal{N}$. They allow to speak of better code, smaller control flow graphs, etc.

The supervisor or control code should guarantee that the result r of the compilation process for input s is:

$$r = g_o^{-1}(MIN_{r' \in D_o}\ \{\ g_o(r') \mid (\_, \ldots, r', \ldots, \_) = (h_1 \circ \ldots \circ h_k)\ (\_, \ldots, s, \ldots, \_)\})$$



produce ---------->
use ———————>
modify .....>

The symbol CF denotes the ConstFold engine and CP the ConstProp engine.

Figure 1: Engine Dependence Graph of our Example

The $h_1, \ldots, h_k$ are a sequence of engine activations (*reasonable activation orders*). The sequence is not allowed to violate the *true* dependence relation. Furthermore the number k should be minimal for the compilers to be efficient, i.e. fast. There exist some heuristics that helps to decrease the length of the sequence (k). The following section presents a method to specify the heuristics as well as an application to our example.

4 Heuristics

The system part controlling the activation of engines is called the *supervisor*. In our approach, the supervisor is generated from specifications. In the given example, the supervisor may select between several possible engines. The ordering between a constant folder (*ConstFold*) and the tree simplifier (*TreeSimpl*) is not clear from the dependences. Both of them may modify the syntax tree. The compiler writer knows that a constant folder should work before the tree simplifier. For example, if one transformation rule of the tree simplifier describes the pruning of a constant conditional **if true then** s_1 **else** s_2 **fi** into s_1, then the constant folder may produce such a conditional by evaluating a constant expression. The supervisor generator should allow the compiler integrator to specify such hints. These hints are normally called heuristics and are dependent on the incorporated engines. We introduce a *priority* relation between engines. The semantics of an element (e_1, e_2) is as follows; when some engines are waiting at a data structure, the selection is according to the priority relation; a minimal element of that relation is chosen, if one exists. Otherwise an engine is chosen nondeterministically. For our example we give the *priority* relation[1]:
$ConstFold \overset{\text{priority}}{\longrightarrow} TreeSimpl$, $ConstFold \overset{\text{priority}}{\longrightarrow} Live$ and $Live \overset{\text{priority}}{\longrightarrow} TreeSimpl$. Any sequence of engine activations $(h_1, \ldots, h_k)$ describing the runtime behaviour of the compiler should also not violate the *priority* relation to guarantee the efficiency.

The *priority* relation is used by the generator in the following way. In any static case where the generator has to make a choice between two engines it inspects the priorities to select one first, in any dynamic case the generator inserts code that does the desired choice. That means for our example, if there are choice points in the supervisor generator, it generates code such that the constants are folded first, then the dead variables are eliminated and afterwards the tree transformations are done.

5 The Concept of Distributed Supervision

First we want to define what are the two tasks of supervision. The more important task is to guarantee the correct cooperation of engines. We assume the correctness

[1] an element (e_1, e_2) of the *priority* relation is drawn $e_1 \overset{\text{priority}}{\longrightarrow} e_2$)

of the incorporated engines (*correct algorithms*). The second one is to increase the compilation speed.

To achieve these two goals, we use the two relations defined in the previous section. The *engine dependence graph* expresses the correctness constraints and the *priority* relation contains the heuristics for the efficiency.

In general, the code for the scheduling of such a pool of engines can be distributed among the engines or it can be kept and executed in a centralized way. The distribution of the code results in more efficient compilers because the distribution of the control code over the engines results in less shared data structures and therefore in less synchronization code and overhead at runtime.

In distributed supervisors, any engine (i.e. the implementation of a compiler task) will be surrounded by an envelope doing the synchronization with other engines on the compiler data structures. That envelop has two major tasks. First, it checks whether an engine can be exccuted; it implements access to all needed structures. Secondly the supervisor guarantees the *data consistency*; we say a data structure is *consistent*, if the data structures it is produced from have not changed after d was computed and if these are also consistent. This definition of consistency covers two different tasks;

- Correctness
  Engines produce versions of data structures from versions of other data structures. If an engine e accesses two data structures d_1, d_2 that are (transitively) dependent on a same structure d then the version number on which d_1 and d_2 are based has to be the same.

- Efficiency
  If an engine e want to access a data structure d and it is known to the system that d may change (because a modifier has announced the access to d or an engine works on a data structure d is dependent on) then the run of e should be delayed until the work has been finished on d.

The correctness requires an accounting of data structure versions. An engine is only allowed to be activated if the data structures it is based on, are consistent. We define a consistency check in section 7, which solves both tasks. This definition has the nice property that it reduces the number of activations, i.e. it increases the speed of the compiler.

The code of an engine will remain unchanged by the synchronization generator. Therefore, the engine can be replaced by another engine doing the same job but using a different algorithm, if the functionality is specified with our method and the algorithm is capable to support additional informations (explained in the next section).

The supervisor envelope of an engine consists of a declaration of control data structures and control code. The control data structures can only be accessed by the control code; they are invisible from the engine.

We summarize our intention of a distributed supervisor;

A distributed supervisor consists of control code and data that is distributed among the compiler engines to control the compilation process. The control code and data structures are invisible from the engines. The supervisor generator is not allowed to change the code of the supervised engines.

We now explain how distributed supervision works (Figure 2 shows the generation process). The tasks that have to be solved for the correctness of the compilation process can be split into the subtasks:

- access to the data structures
- consistency check (correctness part)
- scheduling

In the supervisor code as described in this paper, we use the following abstractions: First we need a mechanism triggering the *coming into existence* of a data structure; e.g. the control flow graph constructor cannot begin to work until the parser has built the syntax tree. The macro *Wait_on_Existence(d)* stands for a code sequence that performs this delay of an engine; it implements the waiting[2] until (the first version of) the data structure d is available. In sequential compilers, this is normally guaranteed by the total ordering of engine calls but in the case that all engines are started concurrently we need such a mechanism.

Secondly, we need macros doing the synchronization on the different data structures. They are called *Get_(Read—Modify—Write)_Access* in the abstract code and implement the necessary synchronization. To that end, access rights to data structures are passed between engines. An engine tries to obtain the access rights of the needed data structures in the order *write*, *modify* and *use* that is explained together with the rescheduling of engines at the end of this section.

If an engine has obtained the access rights to some data structures it has to check the consistency of those structures as mentioned above. The macros *consistency* are doing that.

Before and after a run of an engine, some administrative work has to be done, updating local informations including the versioning of data structures. This is done by *Store_Versions* before and by *Update_Versions* after the activation of that engine.

After the activation the access rights to the touched data structures have to be released by the engine. The macro *Release_Access(d)* does the administrative work on the data structure d; that includes the scheduling of processes waiting on that structure.

For the detection of the termination of the compilation (i.e. the convergence; see section 6) we need additional mechanisms. The macro *Terminate* is used signaling local

[2]the waiting concept (busy, suspend–resume, etc.) is host architecture dependent, i.e. a parameter of the generator.

convergence for this engine, i.e. the last computation delivered the same results as the previous one.

After a run of an engine it has to wait whether the activation of other engines requires an additional activation; *Wait_on_Change(d)* suspends the computation of that engine until a change occurs in used or modified data structures.

New_Run signals to the termination detection, that a data structure it wants to use or modify has changed and that therefore the engine must be run again.

For the efficiency of the resulting compiler the optimization of the general synchronization mechanism as well as the use of the *priority* relation is important. It will influence the implementations of the *Get–Access* and the *Release* macros. Section 9 contains an improvement on the number of consistency checks.

For every engine e the control data structures consist of:

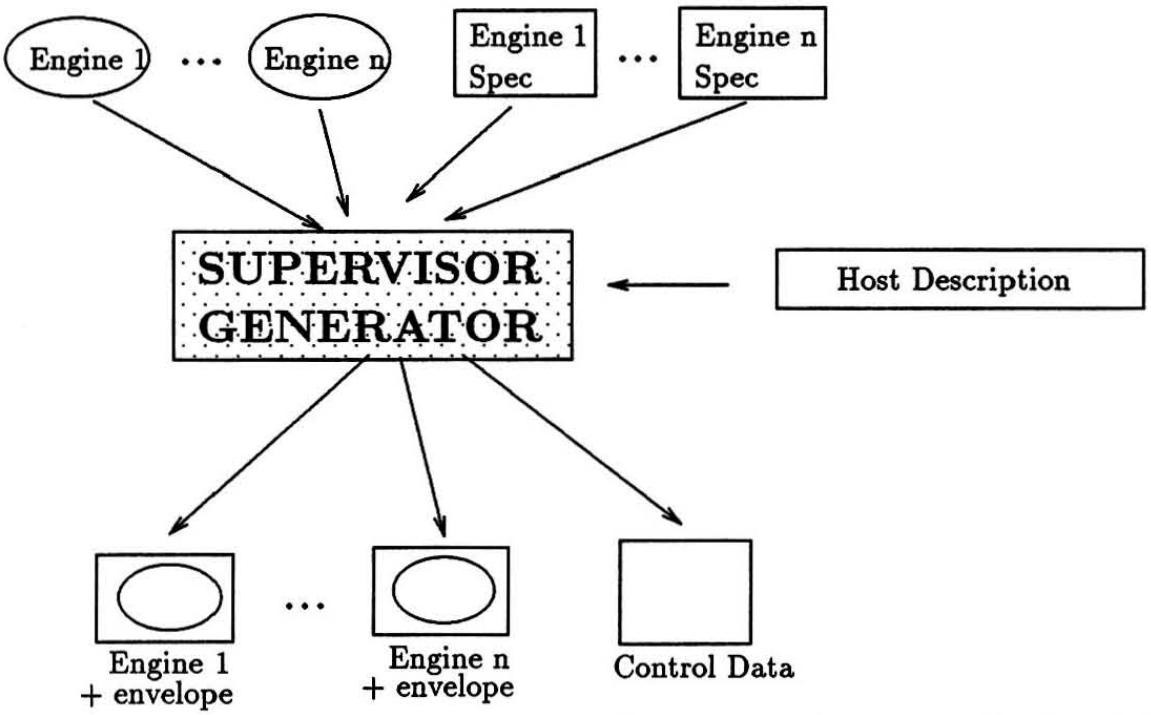
- one flag for any produced or modified structure d, indicating whether the current information has been changed since the last run of that engine: *e_d_changed*. These variables have to be set by the engine, i.e. the requirements on the engine.
- a set of (integer valued) variables (for any used structure d) that stores the versions of the information used in the last activation: *e_d_version*.
- a set of identifiers of shared memory segments containing the data structures that are accessed by that engine: *e_d_shared*.

For any data structure d we have an additional shared variable *d_version* representing the current version number of that data structure and a shared segment *d_shared* that contains the data structure. We assume a mechanism *Map* that links a segment identifier to a segment. It has to be supported by the host architecture.

The generated supervisor guarantees that the scheduling of engine e does not violate the *true* relation. Engines contained in PU(e) are scheduled before e, and in competition for the access to data structures the set PUS(e) instead of PU(e) is used. The macros $USE(e)$, $MOD(e)$ and $PRO(e)$ are denotations for the sets of used, modified and produced data structures of an engine e.

The code of the distributed supervisor, produced from the *engine dependence graph* and the *priority* relation, for an engine e is given below in an abstract (imperative) language; it is the envelope for an engine e handling the synchronization with other engines. The distributed supervisor as entity consists of all such distributed parts together with one start sequence. This start sequence contains the code installing the engine activations and initializing the operating system interface.

The used macros are explained later on in more detail. The case study in section 12 gives details about a real application.



The supervisor generator takes the engines, the engine specifications and a description of the host architecture and produces the control code (engine envelope) together with the control data. The description of the host architecture contains the synchronization primitives.

Figure 2: The Generation Process of Supervisor Code

We need an additional (data structure) set for the abstract supervision code. The set $USE'(e)$ is defined as the data structures that are only used;

$$USE'(e) = \ USE(e) - MOD(e)$$

```
void synchr_engine_e;
begin
foreach x ∈ USE'(e) ∪ MOD(e) ∪ PRO(e) do
      Map(e_d_shared,d_shared);
end;
foreach x ∈ USE'(e) ∪ MOD(e) do
      Wait_on_Existence(x);
end;
while true do
      foreach x ∈ PRO(e) ∪ MOD(e) do
         e_x_changed = false;
      end;
      had_run = false;
      foreach x ∈ PRO(e) do
            Get_Write_Access (x);
      end;
      foreach x ∈ MOD(e) do
            Get_Modify_Access (x);
      end;
      foreach x ∈ USE'(e) do
            Get_Read_Access (x);
      end;
      foreach x ∈ USE'(e) ∪ MOD(e) do
            if ⋁ (not consistent(x)) goto release; fi;
      end;
      had_run = true;
      Store_Versions(e);
      let (d_1, ..., d_n) = PRO(e) ∪ MOD(e) in
         (e_d_1_changed, ..., e_d_n_changed) = activate_engine e;
         Update_Versions(e_d_1_changed, ..., e_d_n_changed);
      tel;
release :
      foreach x ∈ USE'(e) ∪ MOD(e) ∪ PRO(e) do
         Release_Access(x);
      end;
      if had_run then Terminate; fi;
      foreach x ∈ USE'(e) ∪ MOD(e)
            Wait_on_Change(x);
       end;
      if had_run then New_Run; fi;
end;
```

One task is still unsolved namely the rescheduling of processes. The problem arises if some engines have to wait on a data structure. The selection of these engines that

are allowed to continue works according to the following strategy. There exist three possible access methods: read, modify and write. We introduce the total ordering write < modify < read[3]. That has the following consequence. If there is one producer (or the producer) of that data structure in the waiting queue, let this one proceed. If there is no producer but at least one modifier then let one of these modifiers proceed. If there is no producer and modifier at all then let all the readers proceed. Note that there is no time criterion in it as in usual waiting queues, i.e. they are no queues but sets. The priority relation is used to select the right one in a set of possible choices, i.e. if there are more than one modifier a minimal element of the priority relation between engines is selected. The *priority* relation is also inspected, if there are concurrent calls to the *Get_Write-Modify-Read_Access* functions.

6 Termination Detection

We define the termination of the compilation process as the convergence of the engines on the data structures. In this sense the convergence is a sufficient condition for termination. The consequences of this termination concept are an additional constraint on the incorporated engines. In sequential compilers the convergence can be detected by a single boolean variable, but in the parallel scenario, the global termination detection of parallel processes is more difficult. It can be based on the detections of local convergence of the involved engines. If no engine changes anything, then (global) convergence of the compilation process is reached. The requirement on the compiler engines to compute the boolean flags, indicating whether there is a change in the touched data structure, allows the synchronization compiler to generate that code. However, this is a constraint on the implementation of the involved engines.

For the detection of global convergence, the generator produces a shared variable *#engines* that is initialized with the number of incorporated engines. On that shared variable we only have two (monitor) operations: *Increment* and *Decrement* (by one).

The macro *Terminate* is implemented by the code sequence

```
if not (⋁ e_d_i_changed)
       then Decrement(#engines)
fi
```

The macro *New_Run* can be simply implemented by the *Increment* operation on the variable *#engines*.

An engine using a data structure must be informed that there is a new *version* of that data structure. We need an additional control data structure storing the *versions*. This can be done with any data type having an equality operation and an operation

[3]We call this ordering *natural*, because we have to produce data structures before we can access them.

producing a new element that has never been occurred before. We have chosen the data type *integer* with the usual equality. The other operation is chosen as the increment operation on the integers that has the necessary property.

We now explain in detail how the versioning is done. Any data structure has, as mentioned above, a shared variable counting its current version of that structure (*d_version*). Any engine that uses or modifies that data structure has a variable storing the version of the data structures that has been used in the last activation. Before an engine activation, the macro *Store_Versions* does the local update of the version numbers, i.e.

$$\begin{aligned} \mathit{Store\_Versions}(e) = \quad & \forall\, d \in \mathit{USE}'(e) \cup \mathit{MOD}(e) \\ & \mathit{e\_d\_version} = \mathit{d\_version}; \end{aligned}$$

The access to any shared variable is protected either by a monitor function or a semaphore.

After a successful run the local update is done as follows. The local information is set to the new version numbers, if there have been changes on the data structures.

$$\begin{aligned} & \mathit{Update\_Versions}(\mathit{p\_d_1\_changed}, \ldots, \mathit{p\_d_n\_changed}) = \\ & \qquad \forall\, i \in 1..n \\ & \qquad\quad \textbf{if } \mathit{e\_d_i\_changed} \textbf{ then } \mathit{Increment}(\mathit{d_i\_version}) \textbf{ fi}; \end{aligned}$$

The variable *had_run* is used to distinguish in the part of the code after the label declaration whether there was an activation or only a consistency failure. In the latter case the termination detection has to be suspended until the next activation.

7 Consistency

The consistency check consists of a function call that may call consistency checks of other engines. A version of a data structure d is consistent (as mentioned before), if the data structures it has been produced or updated from have not changed since the last computation and if these data structures are consistent themselves. If $d \in \mathit{PROD}(e)$ then the consistency of d can be computed as follows

$$\mathit{consistency}(d) = \bigwedge_{\substack{z \in \mathrm{MOD(e)} \\ \cup\ \mathrm{USE'(e)}}} (\mathit{e\_z\_version} = \mathit{z\_version} \wedge \mathit{consistency}(z))$$

The code first checks the considered data structure d and afterwards the consistency of the data structures that are used to produce the structures.

8 A Specification Language for Engines

The input language of our engine description compiler was kept as simple as possible abstracting from the underlying host architecture. The specification of an engine includes the three kinds of possible access methods to data structures, namely *produce*, *modify* and *use*. The *modify* access specification implicitly also defines a *use* access. Furthermore it includes a specification method for the *priority* relation and some administrative things like name of the file containing the engine and name of the function performing the algorithm (that is called in the supervision code). We omit the administrative parts for simplicity reasons. As simple as the specification language is, it contains all necessary informations that is needed to generate correct and efficient supervision code for the compiler. The host architecture itself is abstracted by a library mechanism containing the machine dependent things like the implementation of synchronization primitives and the data types for the control data. It cannot be specified in the current version.

A source code optimizer can be specified by the following simple specification. This engine may change the syntax tree looking for some (syntactical) patterns and is more successful in finding such one, if a constant folder or a dead variable eliminator have produced subtrees that are matched by these patterns.

Example: 2 (Specification of the tree_simplifier)

```
ENGINE   TreeSimpl IS
         MODIFIES                Syntax Tree
         PRIORITY LESS THAN      ConstFold,Live
END
```

The Appendix 1 contains the engine specifications of the example 1.

9 Improvements

Several improvements are possible because the generation of the distributed control code has knowledge about all engines and data structures at supervisor generation time.

First we show that we can inspect the data dependence graph and reduce the amount of synchronization work at runtime. The presented consistency check may check a data structure more than once during the same check. The improvement of the consistency check only checks the necessary structures.

The consistency check consists of a local check of the directly accessed data structures and a consistency check of other structures. These may include a check of structures

which are already checked. It can be computed from the engine dependence graph whether that is the case. The consistency check for a data structure d that is in PROD(e) can be reduced to

$$consistency(d) = \bigwedge_{z \in U'(e)} e\_z\_version = z\_version \wedge consistency(z)$$

where $U(e)'$ is the set of used and modified structures on a slightly modified EDG. That EDG' = (V',E') is defined as follows: the vertices are not changed i.e. V' = V and the edge set E' is:

$$\mathrm{E}' = E \setminus \{d \xrightarrow{\text{use}} e \in \mathrm{E} \mid \exists\, d' \xrightarrow{\text{use}} e \in \mathrm{E}\ and\ \exists\, d \xrightarrow{(\text{use}|\text{modify}|\text{produce})^+} d'\ and\ d \neq d'\}$$

We eliminate edges that results in useless synchronization by using the fact that they are already checked by other consistency checks.

10 Conclusion and Future Work

We have presented a new generative approach to the ordering of engines in compilers. For any engine the compiler integrator specifies the input-output behaviour and from these specifications distributed control code and control data are generated. It guarantees the data consistency of the data structures during the compilation process and the optimality of the result with respect to the incorporated engines. The compilation time however might be slightly suboptimal for some host architectures, because we rely on the implementation of the operating system primitives.

The approach is also very interesting from the point of software engineering, because it allows the fast exchange of compiler engines without changing the other engines and the synchronization code manually.

We have synchronized engines on the coarse grain level. The access methods access the data structures as one entity, which is not sufficient for massive parallel computers. We will investigate the granularity of synchronization. We are also working on a formal method for the specification of the data structures.

Future work will concentrate on how to guarantee and detect *termination* of compilation as well as a more sophisticated deadlock prevention.

The incorporation of incremental engines seems to play an important role for the efficiency, i.e. the compilation time, and will be examined further, because they introduce additional constraints on the synchronization code as well as on the engines.

We will also remove the restriction that a data structure can be produced only from one engine by introducing an additonal (priority) relation on engines producing that structure.

11 Related Work

In general the attempts to parallelize compilers can be divided into two main directions. One tries to parallelize algorithms like scanning or parsing and is source language independent. One related work is [7]. The others parallelize existing sequential compilers and are therefore source language and compiler dependent (as example [15]). The work of [11] goes that direction by handling different scopes in parallel and combining the results at the end. In [10], they use a similar appproach but construct a library of synchronizations primitives and modify the source code of the compiler engines by introduction of calls to those synchronization primitives. [16] define a meta language with synchronization primitives that allow to program the synchronization of engines by hand. In early works of [13] and [5], they define the concepts of a *Module Interconnection Language* (MIL). The specification language consists of a similar input-output specification but is dependent on the implementation language of the modules, because it includes its type concept.

Our approach is neither source language nor target language nor compiler dependent, but requires the availability of the compiler engines. Therefore we believe that it is also applicable in other areas than compiler construction.

12 Case Study

We have implemented the proposed generator (in ANSI C) and tested it on a shared memory architecture with eight processors running a UNIX operating system. The supervisor code can be generated either in ANSI C or plain C. We used data structures and operations for control code and data that are supported by the operating system (Sytem V primitives). That decreases the efficiency, because we have an additional synchronization layer, however it results in a portable system.

The engines are mapped to parallel processes and the scheduling of these is done by the operating system of the machine. We used counting semaphores as process communication interface. The implementation of the additonal synchronization primitives is built on top of them.

The synchronization primitives using semaphores only allow a simplified form of multiple readers of data structures but exclusive modify and write access. The scheduling of the data structures is mapped on the normal scheduling of the operating system for the waiting processes; there we have an additonal time constraint due to operating system scheduling, which is in our case first in first out; the waiting processes are managed with queues and the control code is not able to respect (user defined) constraints in selecting one process. In this sense the ordering $produce < modify < read$ is neglected as well as the priority relation. That does'nt results in wrong compilers but precludes further exploitation of parallelism.

The termination detection is done by an additional process checking the shared variable *#engines*. If that variable becomes zero the compiler processes are removed from the machine.

All access function calls of an engine envelope are grouped in a critical section to prevent deadlocks (see [4]).

We used our implementation to generate the supervisor for a superset (22 engines) of engines from our example. They optimize a toy language (subset of PASCAL) on the source level and transform it into a vector dialect of that language.

Our prototype implementation is straightforward but gives strong reasons to believe that a sophisticated implementation will be quite a bit faster.

As an indication for the ease of integration of new engines we may mention that the inclusion of the vectorizer cost only one man day. The time was mainly spent to adapt the (local) data structures of the vectorizer to the parallel machine.

Summarizing the facts we believe that we gain two things using our method:

- compiler become simpler to maintain and to port to other architectures; that is the software engineering aspect
- compiler are faster; that is the efficiency aspect

A Example Specification

The following specification describes the structure of the compiler from example 1.

ENGINE Scan IS
USES Character Stream
PRODUCES Token Stream
END

ENGINE Parse IS
USES Token Stream
PRODUCES Syntax Tree
END

ENGINE ControlFlow IS
USES Syntax Tree
PRODUCES Control Flow Graph
END

ENGINE DefUse IS
USES Syntax Tree
PRODUCES Def-Use Sets
END

ENGINE CP IS
USES Def-Use Sets,Control Flow Graph,
Mod-Var Sets
PRODUCES Constant Sets
END

ENGINE CF IS
USES Control Flow Graph, Constant Sets
MODIFIES Syntax Tree
END

ENGINE ModVars IS
USE Syntax Tree, Def-Use Sets
PRODUCES Mod-Var Sets
END

ENGINE Live IS
USES Control Flow Graph
MODIFIES Syntax Tree
PRIORITY LESS THAN CF
END

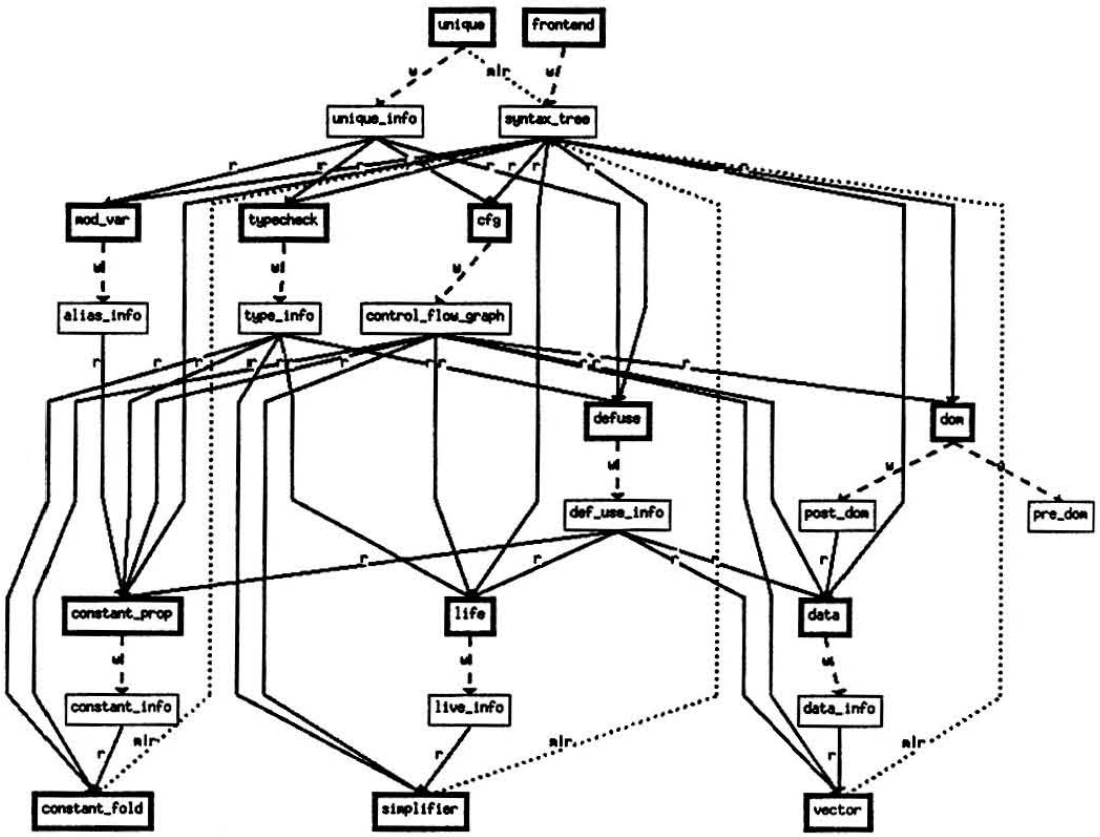
ENGINE TreeSimpl IS
MODIFIES Syntax Tree
PRIORITY LESS THAN Live,CF
END

B Time Measurements

In general, time measurements on parallel machines are more complicated than in the sequential case, because they are dependent on the task scheduling of the operating system. We have to measure the absolute time of the compilation process, but this is load dependent. The load of the system is not the same in the run of the parallel and the sequential compiler; we cannot compute the exact speedup. For these reasons we added the system load to the time table. It can be seen that the machine was heavily overloaded. The structure of the parallel compiler[4] we measured is shown in figure 3. We measured absolute time in seconds.

System-Load	Parallel	Sequential	Speedup
13.46	1050	1720	1.64
15.62	1574	2071	1.31
21.62	1870	2969	1.58

[4]extension of the example



The bold boxes denote engines, the solid one are for data structures.

Figure 3: The (simplified) engine dependence graph of the case study compiler; only the important engines are shown.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] F.E. Allen. Control Flow Analysis. In *SIGPLAN*, volume 7, pages 1–19, 1970. 5.

[3] D. Callahan, K.D. Cooper, K.W. Kennedy, and L.M. Torcon. Interprocedural Constant Propagtion. In *SIGPLAN 86 Symposium on Compiler Construction*, pages 152–161, 1986. 21.

[4] H.M. Deitel. *An Introduction to Operating Systems.* Addison Wesley, 1984.

[5] Frank DeRemer and Hans Kron. Programming-in-the-large versus Programming-in-the-small. *IEEE*, Nov 1976.

[6] M.S. Hecht. *Flow Analysis of Computer Programs.* North Holland, New York, 1977.

[7] Richard Marion Schell (Jr). *Methods for constructing parallel compilers for use in a multiprocessor environment.* PhD thesis, Urbana Illinois, 1979.

[8] J.B. Kam and J.D. Ullman. Monotone Data Flow Analysis Frameworks. In *Acta Informatica*, volume 7, pages 309–317, 1970.

[9] Monica S. Lam and Martin C. Rinard. Coarse–Grain Parallel Programming in Jade. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Williamsburg, Virginia, April 21-24*, volume 26, 1991.

[10] D. Scales, M. Rinard, M. Lam, and J. Anderson. Hierachical Concurrency in Jade. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing. 4th International Workshop, Santa Clara, California, USA, August 7-9, 1991.* Springer-Verlag, LNCS 589, 1992.

[11] V. Seshadri, D.B. Wortman, M.D. Junkin, S.Weber, C.P. Yu, and I. Small. Semantic Analysis in a Concurrent Compiler. In *Proceedings of the ACM Sigplan '88 Conference*, 1988.

[12] David B. Skillicorn and David T. Barnard. Parallel Compilation: A Status Report. Technical report, Queens's University, Kingston, Ontario, 1990.

[13] Walter F. Tichy. Software Development Control Based on Module Interconnection. In *Proc. of the 4th International Conference on Software Engineering*, Sep 1979.

[14] Walter F. Tichy. Programming-in-the-Large: Past, Present and Future. *Communications of the ACM*, 1992.

[15] M.T. Vandevoorde. *Parallel Compilation on a Tightly Coupled Multiprocessor.* PhD thesis, digital, Systems Research Center, 1988.

[16] Gio Wiederhold, Peter Wegner, and Stefano Ceri. Toward MEGA Programming. *Communications of the ACM*, 35(11), Nov 1992.

[17] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau.* Springer Verlag, 1992.