Cosy Compiler Phase Embedding with the CoSy Compiler Model*

Martin Alt[1] Uwe Aßmann[2] Hans van Someren[3]

[1] Universität des Saarlandes, alt@cs.uni-sb.de
[2] Universität Karlsruhe IPD, assmann@ira.uka.de
[3] ACE Associated Computer Experts bv, Amsterdam, hvs@ace.nl

Abstract. In this article we introduce a novel model for compilation and compiler construction, the CoSy(COmpiler SYstem) model. CoSy provides a framework for flexible combination and embedding of compiler phases — called engines in the sequel — such that the construction of parallel and (inter-procedural) optimizing compilers is facilitated. In CoSy a compiler writer may program some phase in a target language and embed it transparently — without source code changes — into different compiler contexts, such as with alternative phase order, speculative evaluation[4], parallel evaluation, and generate-and-test evaluation. Compilers constructed with CoSy can be tuned for different host systems (the system the compiler runs on, not the system it produces code for) and are transparently scalable for (shared memory) multiprocessor host configurations.
To achieve this, CoSy provides an engine description language (EDL) which allows to describe the control flow and interaction of compiler engines. A novel structure definition language (fSDL) is introduced for the specification of data, access side effects, and visibility control. A configuration description (CCL) is used to tune the embedding of the engines to the host system characteristics and limitations.
In order to guarantee transparent embedding, CoSy introduces for each engine a logical view on the common intermediate representation. CoSy generates code that maps these views onto the physical representation, code that helps the programmer to manipulate it, as well as code that schedules the interaction of the engines.
The proposed model of compilation does not depend on source language, programming language, or host architecture. It is not restricted to compiler construction and may have applications in a wide area of software engineering.

1 Motivation for CoSy

One of the major aims in compiler construction is to improve the efficiency of the generated code by extensive optimization. While this has become more

* This work was supported in part by ESPRIT project No. 5399 COMPARE

[4] the concurrent evaluation according to different strategies with subsequent selection of the best one

and more important since modern processors (RISC, superscalar, multi-pipeline) have appeared, which is the best (or even a good) strategy for optimization, is still unclear.

An optimizer writer has to choose between three strategies. The first is to use an optimal implementation of the optimization method. This strategy is unrealistic because most optimizations are known to be NP-hard, and thus can be applied only for very small parts of code (compare the superoptimizer approach [GK92] or the work on peephole optimizer generation).

Therefore, usually the second possibility is chosen, which is to use heuristics. An additional reason for this is that not all information may be available at a certain point in the compilation; we may have to make weak approximations about the effects of subsequent compiler phases. This may result in an algorithm with very unpleasant results in certain situations. Instead of making assumptions about effects of some phases, one would like to inspect the effects by letting those phases run, i.e. one would like to weaken the classical sequential phase ordering.

The third possibility is to apply several heuristics or a heuristic with several settings concurrently, evaluate their results and choose the best result for further processing. In essence this means that competition between different algorithms or different parameterizations is performed. This becomes more and more applicable because shared memory multiprocessors become common, and one would like to make use of their parallel processing capability. However, the use of this technique causes a lot of implementation overhead. Not only the comparison and assessment of the results have to be implemented but also specialized data structures and algorithms which take the speculative evaluations into account.

The CoSy model of compilation has been developed: to weaken the sequential phase ordering and to enable parallel competitive evaluation. Its aim is to facilitate the development of fast compilers generating efficient code. It provides means to embed handwritten or generated compiler phases as black boxes into different compiler contexts without source code changes. It can be seen as a compiler construction toolbox, that gives us the possibility to test compiler structures and select the best one. And it will also enable us to utilize optimization by speculative evaluation, generate-and-test evaluation, parallel evaluation, and alternative orderings.

1.1 Potential of CoSy

A lot of optimization problems benefit from the capabilities of CoSy; in the following we will give some examples.

Several problems can exploit speculative evaluation. Register allocation can be performed with different numbers of registers for parameter passing, global registers, and basic block local registers. Because register allocation and instruction scheduling influence each other, speculative evaluation can be used to find a better code ordering [Bra91]. Different register allocation methods can be applied competitively, and a machine simulator or a performance prediction tool engine

can be used to select the best version (this also means that the simulator or performance prediction engine is part of the compiler context). Currently there are a number of different algorithms for pointer alias analysis; such analysis is the basis for optimization on programs with pointers. Because it is difficult to judge in advance which of them yields the best result they can be run competitively and by combining their results better information can be obtained. There are many more examples, but the central issue is: how can we avoid reprogramming engines when we embed them in a different context? How can we experiment with different parameterizations of the algorithms and make use of competitive evaluation?

With speculative evaluation there are alternative algorithms to produce the competing versions. We can also use the generate-and-test method, employing one algorithm which generates several alternative versions, one after the other. Generate-and-test is applicable when the range of the parameters we want to manipulate is intractably large. As an example regard the register allocation algorithm of Chaitin [Cha82]. It has about two parameters we can tune, the first is which registers are deleted from the interference graph in a round and the second is which registers must be spilled if the graph cannot be colored. We can easily imagine a register allocator that generates several of these allocations and have again a selector engine that measures register life times to select the best version. We may stop the production of new versions if we think we have reached a local optimum. Generate-and-test, however, needs special prerequisites in the engines. Is it possible to avoid reprogramming of engines and embed them transparently into such a context?

Another parameter that influences the optimization results is how the optimization engines are ordered. Because certain optimizations enable or disable others (see [WS90]), the optimizer writer should be given a means to embed an engine into several ordering contexts, e.g. into loops that run until no optimization can be applied anymore (exhaustive transformation); or into loops whose iteration number is configured by the machine. Of course exhaustive transformation can be implemented with support of the algorithm to indicate what changed, but this may slow down an engine when it is embedded in a non-loop context. How can we automate the detection of such changes and achieve transparent embedding of engines into alterable orderings?

One method to speed up an algorithm is making a parallel version for it. It is often argued that this is of no use in a compiler because there are a lot of dependencies among the engines. While this is true in general there are certain optimizations which don't have internal dependencies, e.g. when some analyses are done on procedures in a data parallel way [LR92]. For these cases we would like to give the programmer an easy method to embed his engine into a parallel context. However, this normally requires rewriting of a large amount of code because synchronization primitives have to be inserted. How can the transparent embedding of an engine into a parallel context be achieved?

Besides that CoSy answers all these questions a further advantage is that it uses a clear specification methodology for interfaces between the engines to

enhance the modularity and the maintainability of the generated systems. We are able to replace engines by new releases with a small amount of time and system resources. Because side effects of engines are known precisely, the effects of such an exchange can be estimated and lead at most to a regeneration of the system. This also enhances reusability and interoperability between several compilation systems. It is easy to reuse some engine from an other system, also in object format. The clear specification allows to generate efficient sequential compilers for uni-processor systems.

The aim of this overview article is to explain how CoSy solves flexible engine embedding transparently. In essence this is achieved by a separation of interaction of engines, the engines themselves, and the access of the engines to the common data. While engines are user-provided (hand-written or generated), interaction of engines and access of the engines to data are generated by CoSy and depend on the embedding of the engine. For this CoSy introduces three languages that work together. EDL (engine description language) describes the interaction of the engines and is used for generation of control flow embedding, i.e. supervising code. fSDL (full structure definition language) is used to describe the view of an engine including its side effects so that the access functions to the data can be generated according to the embedding as well as the data structure definitions. And CCL (CoSy configuration language) makes it possible to adapt the configuration of the generated compiler to the particular characteristics and limitations of the host system for efficiency reasons.

2 Mechanisms for flexible engine embedding

In CoSy we use the following compilation model. A set of engines work in parallel on global data (common data pool, CDP) so that exchange of information is easy. Engines are synchronized under the control of (generated) supervisor code. Access to the data is — and this is (programming) convention — done using access routines from a generated library and macro package, the data manipulation and control package (DMCP).

In order to achieve flexible and transparent engine embedding CoSy uses the following mechanisms. The interaction of engines and the embedding context of an engine is described by a fixed set of *interaction schemes* from which the supervisor code can be generated. With knowledge of the interaction definition, the DMCP library is generated which guarantees synchronized access. This means that the actual (physical) access method of an engine to its data is determined by the context it runs in. Engines can be clustered together to processes and this information can be used to optimize synchronization for these combined engines.

This may be necessary to tailor a CoSy system to the host on which it runs, i.e. to adapt it to a limited number of processors, processes, semaphores or some amount of shared memory. The following sections will explain these mechanisms in more detail.

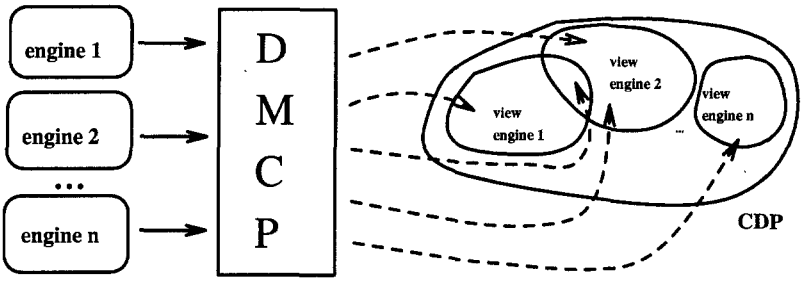

Fig. 1. The CoSy model, engines and the common data pool

2.1 Context specification by interaction schemes

We identified several useful interaction schemes for compilation systems: It is possible to embed an engine into sequence, loop, pipeline, data parallel, speculative, server and generate-and-test contexts.

Sequence and loop interaction model sequential evaluation and iteration over a set of engines. When there are no data dependencies between the activities of some engines, CoSy aims to set up a parallel evaluation of these. Engines in a *loop* may be reactivated for another loop incarnation if a special loop termination engine decides to start the engine sequence again. Such repeated execution normally occurs in a transformation engine which depends on information it modifies (e.g. dead-code elimination together with constant propagation).

Engines which are embedded in a *pipeline* receive an arbitrary number of work units one after the other and overlap their successive operations on them in time. For example, intraprocedural optimization can often be pipelined with the code generator. With *data parallel interaction* as many engines are created as there are work units. Most of the intraprocedural dataflow analysis settings can be mapped to this scheme. Synchronization code for accessing global information can be generated automatically.

A *speculative interaction scheme* embeds a fixed number of engines such that these engines may work without disturbing each other. Each engine gets its own version of the data in which those parts are replicated that would cause dependencies (shadowing). After the run of these engines, a selector engine investigates the results and selects the best. A very interesting application is the concurrent execution of code selectors followed by an assessment engine that chooses the best result.

A *server interaction scheme* allows an engine to call another engine from its algorithmic part, e.g. a constant propagation engine may call an interpreter engine for evaluating constant expressions.

In *optimistic (generate-and-test) interaction* a producer is combined with an experimental engine in such a way, that the producer can create new versions arbitrarily often, and submit them to an experimental engine which can try them

out for feasibility. These versions are tested in parallel, each by a new experimental engine. The results of these experiments are collected and the best one is chosen by a special selector engine. This means that here an arbitrary number of versions can be investigated. Optimistic register allocation with several parameters is a nice example of this.

All interaction scheme specifications are block structured, i.e. an interaction scheme combines some *component engines* to a new one, a *composite engine*, which can be combined again. This results in a hierarchical specification of interaction schemes, the *engine hierarchy*.

This also means that the generation of the supervisor code can be done in a hierarchical way: each interaction scheme results in an *engine stub* that supervises the component engines, and *engine envelopes*, for each component engine one, which provide a standard interface between the stub and the component engines. (See figure 2)

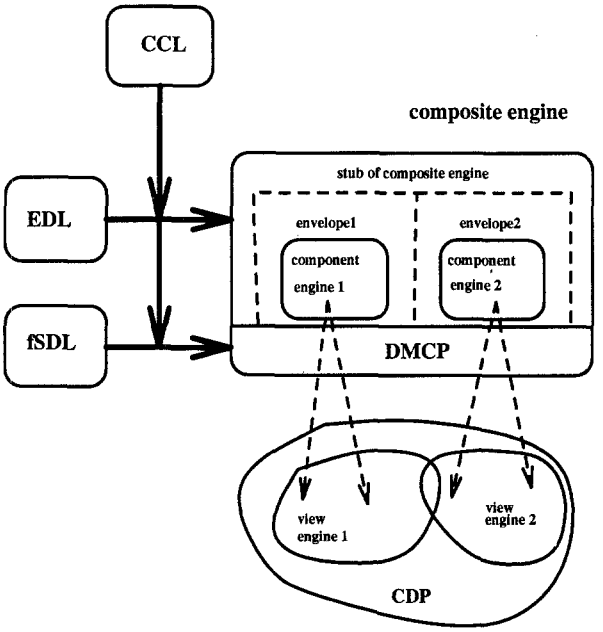

Fig. 2. What CoSy generates and how engines are embedded into the generated code

2.2 Data Structure Definition

We developed a language (fSDL) for the specification of the data structures the engines work on. An engine works on typed memory pieces, called *operators* in

fSDL. Each operator has a fixed number of fields that can be divided into two sets, the annotations and the structure carrying fields that point to other operators. Sets of operators, their fields and properties (about which more below) are collected in so-called *domains*, forming the types in the system. Structural objects consist of operators from a domain with their internal edges. The edges are implicitly drawn by the structural fields. From this data structure specification a variety of functions can be generated, e.g. equality predicates, copy functions etc. The allowed types of fields may be extended by externally defined data structures, called *opaques*; they are the black boxes in the specification. Their use requires that some interfaces have to be explicitly defined such as the equality predicate.

The language has a mechanism for specifying data structures with generic or non-standard functionality, the *functor*. It is a parametrized specification, the parameters are *domains* on which the new structure is based. The result of the instantiation is a data structure (a domain in fSDL) with the added functionality. We will see in the examples through the paper that we import the integer of the machine with its usual operations as *opaque* and the functionality of a list by applying a *functor*.

2.3 Access specification by views and side effects

If we specify engine interactions (control flow) we can not yet guarantee that the system runs correctly, because of the side effects and data dependencies of the engines on the CDP. They can be approximated, if we are able to specify as exactly as possible what an engine does with the CDP. fSDL includes constructs to specify an engine's *logical view*. We are interested in two aspects: which data is touched, and how is it touched. The allowed kind of access of operators and fields can be specified by properties (read, write, new, destroy). Sets of these entities are called *domains*, as mentioned above. The touched data must be reachable from the parameters, that are given to the engine (no global variables). The parameters of engines and fields in operators are typed with *domains*; therefore, domains exactly describe which parts of the CDP are actually touched and how they are touched. The transitive closure over all domains of an engine's parameters make up the *logical view* of the engine.

The following example may illuminate this. Suppose the CDP includes structures of the following form. A module has a procedure list and a procedure has a list of instruction trees. The instructions contain virtual and real register numbers as well as their operands and operations. Figure 3 describes the physical layout of the operators `module`, `procedure`, and `instruction` in the CDP.

For a register allocator this view has to be modified slightly. A register allocator may work only on one procedure, reads the virtual register number, sets the real register number, and modifies the instruction list because it may insert spill code. It never changes existing instruction trees and often only works on one procedure. This set of effects on our example CDP can be described by the domain description from figure 4. The identifiers enclosed in [] denote the access properties which are imposed on the fields in these domains. The logical

```
domain MODULE: {
   module <
      procedure:   LIST(PROC)      // procedure list
   >}
domain PROC : {
   procedure <
      instructions: LIST(INSTR),  // the instruction list
      loops       : LIST(LOOP)    // list of loops, not further specified
   >}
domain INSTR : {
   instruction <
      virtual_register: INT,      // virtual register numbers
      real_register   : INT,      // real register numbers
      operation       : INT,      // enum of operation
      operands        : INSTR     // operands
   >}
```

Fig. 3. Physical layout of data structures

view of the register allocator of our example is also depicted in figure 5. Fields enclosed in the dotted region are visible and read-only while fields in the dashed region are also writable. Other fields are not visible.

```
domain RegisterAlloc: {
   procedure <
      instructions: LIST [WRITE] (RegisterInstruction) [READONLY]
   > }
domain RegisterInstruction: {
   instruction <
      virtual_register: INT [READONLY],
      real_register   : INT [WRITE],
      operation       : INT [READONLY],
      operands        : RegisterInstruction [READONLY]
   >}
```

Fig. 4. Domains of register allocator

The fSDL-compiler (fSDC) only produces those access functions in the DMCP whose fields and effects have been specified in an engine's logical view. This guarantees that engines can neither modify nor read other parts of the CDP. In our example the fSDC produces read and write functions for the fields which implement the list internally and for `real_register`. For the field `loops` no function is generated, for the others only read functions are generated. It is

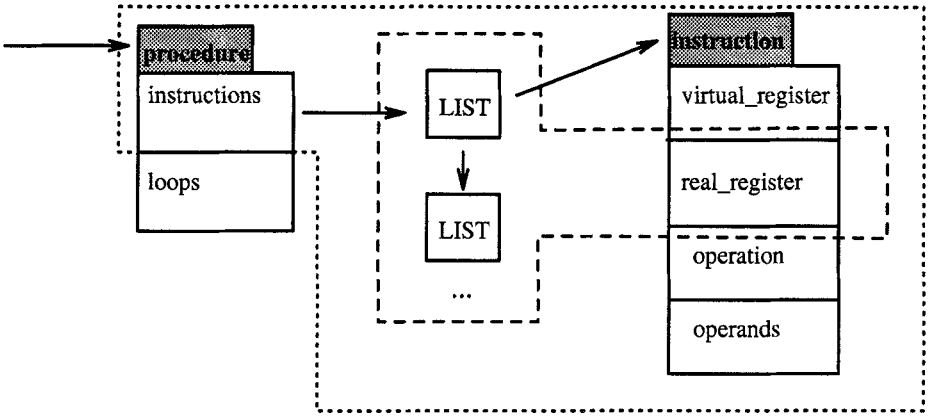

Fig. 5. Domains serve to specify the logical view of an engine

clear that an engine using these domains may not write `virtual_register`, nor replace the list by an other list.

In order to facilitate the definitions of new domains from existing one, fSDL provides a powerful *calculus* over domains that provides inheritance and restrictive operations such that each engine parameter has a related domain which describes exactly the effect of that engine. In essence the following access methods for DMCP functions can be generated.

- Depending on whether the engine reads or writes a field, or allocates or destroys a node, read, write, allocation and free functions are generated.
- If the access to a field does not suffer from a data dependency which may be introduced by the interaction schemes, straight forward access is provided (direct access).
- If there is a data dependency, an access using a serializing protocol is used (serializing access).
- The user may weaken the kind of dependency such that only a locking protocol is used (locking access).
- If the engine is embedded into a speculative or optimistic interaction scheme, the field is replicated (shadowed) and access to the engine's shadow is yielded (shadowed access).

These functions have the same interface but map the engine's logical view to a physical data access which is dependent on the engine embedding (*view mapping*).

All these accesses can either be access functions which are linked-in (linked DMCP) or they can be inlined (inlined DMCP), so that more efficient access is provided. The former approach is more flexible because engines need not be recompiled when they are embedded into other compilers; they just can be linked

with different DMCPs. The latter, an inlined DMCP, provides more efficient access so that more efficient compilers can be built.

2.4 Configuration

Another specification influences the generation of engine stubs (the synchronization code that is generated from interaction schemes) and DMCP access functions. This is the description in CCL, the CoSy configuration language. Despite that logically all engines work in parallel it is possible to reduce the number of parallel units by clustering engines. This has the effect that process invocations to engines collapse to normal procedure calls and synchronization can be optimized or even omitted.

Clustering even serves to achieve a totally sequential compiler if all engines are clustered, or a compiler that consists only of few processes, if a few engines are left non-clustered. Reasonably fast compilers on sequential machines can be configured.

3 Interaction schemes

In this section we will present some examples and show how transparent engine embedding can be used for optimization We will assume that the CDP contains structures as specified in figures 3 and 4. The engine parameters of the interaction schemes that are used to access the physical view of the engine are typed with domain names and with one annotations out of the set IN, INOUT, OUT. The semantics is as follows; an IN handle can not be changed by the engine, an INOUT may, and an OUT handle has to be changed. IN and INOUT handles are initialized, OUT handles not.

3.1 Embedding into parallel context

Suppose we have an intraprocedural register allocator which uses Chaitin's method [Cha82]. How do we organize the process for a list of procedures in a module? There are several possibilities, the easiest one is to run the job in parallel on the procedural level, because there are no dependencies between the register allocations for the procedures. In EDL this would be specified as follows. Note that the parameter of the engine is typed with a domain expression in fSDL which we explained above. If a domains contains exactly one operator or every operator has the same field, we may use the *dot* notation for referencing the fields of that operator.

```
ENGINE CLASS allocate_all(IN m: MODULE) {
   p{} <= m.OPERATOR procedure   // decomposition into work units for
DATAPARALLEL                     // data parallel execution
   registeralloc(p) }
```

This specification declares that so for each procedure an engine **registeralloc** is started in parallel. Of course some mechanism has to be provided that each engine gets its work unit. This is done by a decomposer function which walks the list of procedures and hands over each element (each work unit) to an instance of the **registeralloc** engine. This decomposer can be generated automatically from this specification because the mapping of a list to this set is straightforward or in more complex cases, it can be hand-written. The {} notation denotes the set of work units.

If we don't use clustering this may lead to an enormous amount of scheduling of parallel engines. Instead the register allocator may be embedded into a pipeline, and then only one engine is created which receives the work units one after the other:

```
ENGINE CLASS allocate_all(IN m: MODULE) {
   p<> <= m.OPERATOR procedure   // decomposition into work units
PIPELINE                         // for pipelining
   registeralloc (p) }
```

Note that this does not require the change of a single source code line in the engine. All that is necessary, the generation of work units, the decomposition of the procedure list, the generation of engines and the waiting for completion is generated automatically in the engine stubs, envelopes and access methods.

Of course clustering can be used to coalesce all work into one process if we specify additionally in CCL " **CLUSTER allocate_all** ". This means that **registeralloc** is generated as a sequential subroutine of **allocate_all**, and that the decomposition is just a simple loop over the list of all procedures.

3.2 Embedding into speculative context

In the following we show an example how engines can be embedded into a speculative context. We use as an example competitive register allocation. Suppose the compiler writer has three register allocators working with the three methods [Cha82] [CH84] [HGAM92]. We reuse the view specification from figure 4 for each of these register allocators and also for the speculative composite engine.

```
ENGINE CLASS registeralloc (IN p:RegisterAlloc) {
SPECULATIVE
   chaitin(p)
   chow_henessy(p)
   hendren(p)
SELECT  registeralloc_selector(p) }
```

At runtime the speculative interaction scheme behaves as follows: First the values of the original fields which are to be modified as well as all objects that are reachable from them are copied into shadows (special memory, not accessible from every engine). This means in our case that the field **real_register** is copied into its shadows as well as that the whole instruction list is replicated twice. The replication for any engine-parameter pair can be done in parallel.

Each register allocator works in parallel on its private copy because its generated access functions provide *shadow access*. When all engines are finished the selector determines the best version and copies it back to the original fields, i.e. an unique instruction list is determined again. Then the interaction scheme ends and all other engines can run as if no speculative interaction had taken place.

Of course the selector engine **registeralloc_selector** needs special access functions which give access to all data shadows. If the selection process can be expressed by a mapping of the shadowed structure to the natural numbers and a cost function then the selection can be generated automatically.

Note that **registeralloc** can be substituted in any context by each of the three register allocators without source code change. It is no problem to embed this speculative interaction in the pipelined or data parallel interaction schemes above; the engine hierarchy allows us to do this orthogonally.

3.3 Embedding into alterable engine orderings

In this section we leave the field of register allocation and turn to complex optimization orderings. We show how a simple linear optimization engine can be reconfigured into a complex one with a sophisticated ordering. This is exemplified by the proposal [WS90] which takes several dependencies between the used optimizations into account. First we show how a simple optimizer may be specified in EDL just by concatenating some simple engines. For simplicity we ignore the side effect description here.

```
ENGINE CLASS optimizer (IN p:PROC) {
SEQUENCE
   constant_propagation(p.instructions)
   dead_code_elimination(p.instructions)
   invariant_code_motion(p)
   loop_interchange(p)
   loop_unrolling(p.loops)
   loop_fusion(p.loops)
   strip_mining(p.loops) }
```

In this simple optimizer we perform one optimization after the other. We do not care about a good ordering, i.e. about repetition of engines that are mutually recursively dependent. Such a simple optimizer may be advantageous when we have a sequential machine or when we need quick compilation.

However it is possible to embed the engines in a much more sophisticated way. This following scheme is coded after the engine dependency graph of figure 6 [WS90].

```
ENGINE CLASS optimizer (IN p:PROC) {
SEQUENCE
   constant_propagation(p.instructions)
   complex_opt(p)
   strip_mining(p.loops) }
```

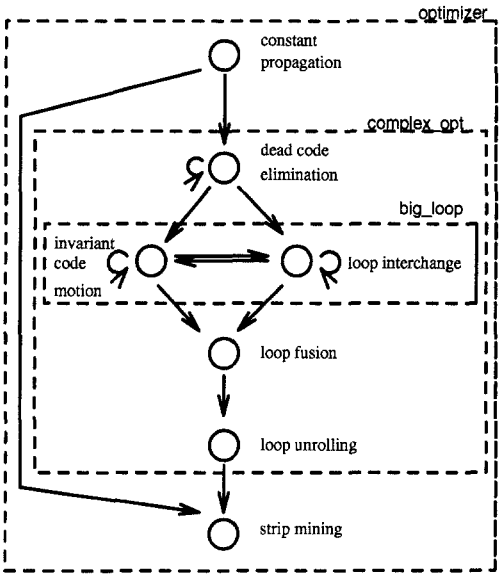

Fig. 6. Dependency graph of optimizations after Whitfield

At the beginning of the optimizer a constant propagator and in the end a strip miner runs. The complex optimizations in the middle are modeled in a sequence interaction scheme:

```
ENGINE CLASS complex_opt (IN p:PROC) {
SEQUENCE
   dead_code_elimination_loop(p.instructions)
   big_loop(p)
   loop_fusion(p.loops)
   loop_unrolling(p.loops) }
```

We only show the complex loop of code motion and loop interchange. This engine is a loop which runs until no further changes occur. Its components are loops over `invariant_code_motion` and `loop_interchange`, respectively.

```
ENGINE CLASS big_loop(IN p:PROC) {
LOOP
EXIT change_detector(p)
     invariant_code_motion_loop(p.instructions)
     loop_interchange_loop(p.loops)}
```

We now embed the engines from the simple optimizer in the complex ordering without source code change. We have to detect transparently whether an engine changes something. Within CoSy the *exit engine* of a loop can be used for this. Initially the `change_detector` engine makes a copy of the data; when it is called

after a loop incarnation, it compares the copy with the new data and, if nothing has changed, terminates the loop.

All used data of the engines is known from the specification of their logical views, so the copying and comparison in the exit engine can be generated automatically from the domain specifications (state tracking). Thus even the exit engine need not be programmed by the user.

3.4 Embedding into generate-and-test context

In this section we show an embedding of the engines **invariant_code_motion** and **loop_interchange** from the previous example in a generate-and-test (optimistic) context. This means that we let a selector decide between an arbitrary number of versions.

Actually in optimistic interaction two selection strategies are combined. First it is decided whether a version is worthwhile by investigating the version alone. Then all versions which have survived are compared and the best one is selected. The semantics of the optimistic interaction scheme can be described by the following pseudo code.

```
do { PRODUCE();
     if (PRESELECT())  TRY();
} while (PRODUCEMORE());
FINALSELECT();
```

This means that a producer engine (PRODUCE) produces new versions of certain optimizations, one after the other, each on separate copies. A second engine (PRESELECT) decides whether the version is worthwhile (pre-selection). TRY is given the version as a shadow and forked off so that it may experiment with the version. In the meanwhile, it is checked by PRODUCEMORE whether PRODUCE should be given another try to produce a version. If so, a next version will be generated and the whole process is repeated. Note that PRESELECT and PRODUCEMORE can run in parallel because they only look at versions and do not modify them. Otherwise, the production stops. The results of the experiment on each version are collected by FINALSELECT which performs the final global selection. This scheme seems to be rather complicated; however, some of these engines may do nothing, in fact. For example, if no pre-selection method is known, there need not be a PRESELECT engine.

```
ENGINE CLASS code_motion_and_loop_interchange(IN p:PROC) {
OPTIMISTIC
  PRODUCE     invariant_code_motion(p)
  PRESELECT   estimate_register_lifetimes(p)
  PRODUCEMORE threshold(p)
  TRY         loop_interchange(p)
  FINALSELECT static_code_analysis(p) }
```

Our example performs as follows. First invariant code motion is done in the PRODUCE engine. As pre-selection criterion we take the amount of register

lifetimes which results from the code motion. Thus we may withdraw already in `estimate_register_lifetimes` some useless versions. Then the TRY engine `loop_interchange` needs to work only on "better" versions. To decide whether the production should be stopped, engine `threshold` could just contain a counter which is increased and compared to a threshold number. In the end, engine `static_code_analysis` evaluates the information of a frequency analysis and estimates the cost of the resulting code versions.

Again the embedding of the engines does not require to change them. Because the side effects are known precisely a lot of work can be done in parallel.

4 Related work

The attempts to parallelize compilers can be divided into two main directions. One tries to parallelize algorithms like scanning or parsing and is language independent. The other others parallelize existing compilers and are therefore source– and target language dependent. In contrast, CoSy will give a very general mechanism (programming languages) for compiler construction.

There have not been many attempts to design languages neither for module re-embedding nor for side effect descriptions. FX [JG90] and JADE [LR91] contain basic principles for side effect description but fSDL goes further because it allows the convenient combination of these descriptions via its domain calculus. JADE also provides a mechanism to synchronize tasks over a shared memory, however its side effect descriptions are mixed with source code so that the re-embedding of a part of the system may not be as easy as with CoSy. CoSy also allows much more kinds of embedding than JADE, and allows flexible configuration and non-deterministic execution.

In early works of [Tic79] and [DK76], they define the concepts of a Module Interconnection Language. This specification language consists of simple input-output specification but is dependent on the implementation language of the modules, because it includes its type concept.

In many aspects CoSy has inherited from IDL [NNGS89]. However, IDL has a somewhat different compilation model: processes communicate via channels and not via a shared common data pool. Therefore IDL requires processes to communicate via external instances like files. Also processes execute sequentially. CoSy allows for much faster communication among engines and still allows correct, also parallel evaluation, because of its powerful data and side effect description mechanism. Furthermore, CoSy allows speculative and optimistic interaction. Therefore the CoSy model will be a major step forward in the reuse of compiler engines and flexible compiler construction.

5 Conclusion

We have shown how the CoSy compiler model can be used to embed compiler engines flexibly into a lot of different compilation and optimization contexts.

CoSy enables the compiler writer to reuse engines, write scalable, parallel, and portable compilers, and tune the performance of his system by exchanging interaction schemes or manipulating the configuration of the system. This all is possible because CoSy provides novel description mechanisms for data structures, side effect descriptions, as well as engine interactions.

For the first time speculative, optimistic, parallel evaluation as well as alterable engines orderings are provided in an orthogonal and unified way for compiler construction. CoSy enables cosy engine embedding: this will facilitate construction of parallel and (inter-procedural) optimizing compilers.

References

[Bra91] D. G. Bradlee. *Retargetable Instruction Scheduling for Pipelined Processors.* PhD thesis, University of Washington, 1991.

[CH84] F.C. Chow and J.L. Hennessy. Register allocation by priority based coloring. In *Proceedings of the ACM SIGPLAN Symp. on Compiler Construction*, June 1984.

[Cha82] G.J. Chaitin. Register allocation and spilling via graph coloring. In *SIGPLAN Conference on Programming Language Design and Implementation*, June 1982.

[DK76] Frank DeRemer and Hans Kron. Programming-in-the-large versus Programming-in-the-small. *IEEE*, Nov 1976.

[GK92] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 341–352. ACM, June 1992.

[HGAM92] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A register allocation framework based on hierrchical cyclic interval graphs. In U. Kastens and P. Pfahler, editors, *CC 92, 4th International Conference on Compiler Construction, LNCS 641*, pages 176–191, 1992.

[JG90] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *ACM Conference on Principles of Programming Languages (POPL)*, 1990.

[LR91] M. S. Lam and M. C. Rinard. Coarse-grain parallel programming in Jade. In *ACM Conference on Principles and Practice of Parallel Processing (PPOPP)*, pages 94–105. Computer Systems Laboratory, Stanford University, 1991.

[LR92] Y-f. Lee and B. Ryder. A comprehensive approach to parallel data flow analysis. In H. Zima, editor, *Workshop on compilers for parallel computers.* Austrian Center for Parallel Computation, 1992.

[NNGS89] J.R. Nestor, J.M. Newcomer, P. Giannini, and D.L. Stone. *IDL:The Language and its Implementation.* Prentice Hall, Englewood Cliffs, 1989.

[Tic79] Walter F. Tichy. Software Development Control Based on Module Interconnection. In *Proc. of the 4th International Conference on Software Engineering*, Sep 1979.

[WS90] D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. In *ACM Conference on Principles and Practice of Parallel Programming (PPOPP)*, 1990.