Firm,
an Intermediate Language for Compiler Research

Interner Bericht 2005-8

Michael Beck, Boris Boesler, Rubino Geiß, and Götz Lindenmaier

Institut für Programmstrukturen und Datenorganisation
Fakultät für Informatik
Universität Karlsruhe
ISSN 1432-7864
{beck|boesler|rubino|goetz}@ipd.info.uni-karlsruhe.de

March 9, 2005

Abstract

State of the art compiler intermediate representations incorporate SSA data dependencies in a graph based manner. We present the intermediate representation Firm, which extends the functional stores of Steensgard and introduces a novel representation of exceptions. Firm offers a high-level representation of the type hierarchy and object-oriented features, which makes it exceptional suitable for analysing and optimizing of strongly typed languages. The construction interface automates value numbering and the generation of SSA typical Phi operations. Firm comes with a full blown range of standard optimizations and analyses. In the paper we show that Firm requires 53% less operationss and 80% less Phi operations than the SSA representation of the gcc compiler.

0.1 Introduction

Optimizing compilers should translate source programs into effective target code, whereas compiler writers want a compiler that is easy to engineer and maintain. To achieve these opposed goals compilers utilize intermediate representations (IRs) as a useful abstraction. We research on compiler optimizations for full fletched programming languages, focusing on object oriented languages. For our research in back end construction we compile imperative languages. As a basis of our research, we need a high quality intermediate representation. We present Firm, an IR that bridges the whole gap between source language and target architecture, using a coherent set of operations.

Firm is a graph based representation extending [Cli95] in two directions: Firstly, we define a novel representation of exceptions in SSA form. Secondly, inter procedural data flow analyses like alias or heap analyses profit from extensions such as an inter procedural representation, an integrated type representation and an extension of Steensgards functional stores.

0.1.1 Our Requirements to an IR

To be comparable, research compilers must generate competitive code, therefore requiring an implementation of standard optimizations typically based on *data flow analysis*. This is best achieved with SSA-form. Further an IR should represent analysis information permanently so that different phases that use the same kind of information do not need to recompute it.

Especially for contemporary languages the IR must represent *exceptions*. Typical IRs either omit exceptions, or their representation causes a severe overhead on data flow analyses. Often they restrict optimizations possibilities because they introduce very strict dependencies. Therefore the IR must represent the required observable state at an exception explicitly to unveil all allowed optimization possibilities.

Pointer analyses are crucial for optimizing object oriented programs. For pointer analyses it is important to identify memory locations where values are stored. Pointer analyses must recognize field accesses to distinguish the content of different fields. Representation of field accesses by address calculations complicate their analysis. To analyse a given Load, a pointer analyses must efficiently navigate in the IR to the corresponding Stores.

Pointer analyses are performed *inter-procedural*. The IR should support inter-procedural analyses by supplying the basic infrastructure. The IR must support a mechanism to represent and to distinguish between different contexts during inter-procedural analyses.

0.1.2 Firm

We developed Firm, an IR to represent common object-oriented and imperative languages. Firm is in *Static-Single-Assignment* form (SSA) [AWZ88, RWZ88], the current state-of-the-art for IRs. Furthermore, it is a fully graph based representation and to our knowledge the only IR in SSA form designed as stand alone representation. This graph represents only dependencies made up by such information as control- and data flow. It leaves out all unimportant information, as sequenzialization of expressions in a basic block. E.g., this allows to express discovered instruction level parallelism (ILP) in the IR and increase it over the compiler phases.

Firm introduces a new representation for exceptions expressing them only in terms of data and control flow. This allows for fine grain adaption of exception context handling to the requirements of the translated source language. The effects of exceptions are fully transparent to optimizations and data flow analyses. They implicitly define a common state variable which sequentializes them in the representation with other operations changing the observable state of a program.

The uniform representation of types contains all information necessary to delay building type mechanisms as explicit type casts and polymorphic method calls. It fully abstracts from source language mechanisms, as naming rules that specify which methods overwrite each other. The tight coupling to the program representation along with high level operations for type dependent mechanisms as dynamic method calls allows straight forward interpretation of these.

We are the first IR to implement *explicit dependency graphs* [Tra01], an extension to the functional stores proposed by [Ste95]. These allow for a better representation of independent stores and more simple analyses of pointer values. This is further supported by the straight forward representation of inter procedural dependencies. Firm is the first IR to extend SSA form to the call graph, explicitly linking formal and actual parameters using special ϕ-nodes.

We present our implementation of Firm. This includes a set of standard optimizations and analyses, as common subexpression elimination or dominator analysis. It defines clean and documented interfaces to construct and access the representation. The SSA construction is automated and it supplies support for inter-procedural analyses.

In Section 0.2 we compare Firm to other IRs in research and commodity compilers. Section 0.3 introduces Firm and explains its advantages in detail. In Section 0.4 we list our applications of Firm. Section 0.5 compares the complexity of Firm with another SSA implementation. Section 0.6 concludes.

0.2 Related Work

We have not yet encountered an industrial strength compiler that uses a pure, graph based IR in SSA form. All IRs we studied distinguish statements and expressions, or are an assembler-like instruction list, sometimes overlaid with a control flow graph. I.e., they resemble the structure of source or target languages. Many compute an SSA representation on top of this representation for certain optimizations, but none represent the information making up the semantics of the program in their basic structure, which are control and data dependencies. Statement/expression IRs represent exceptions on their source language abstraction. That operations reach the same handler is expressed by embracing them with dedicated constructs. Low IRs contain the explicit exception handling mechanisms. No IR includes a representation of inter procedural dependencies.

0.2.1 IRs in Industrial Strength Compilers

Gcc [Sta02] can translate various programming languages. It utilizes two different IRs. The so called tree IR is a high level representation. It is a tree based, syntax tree like representation. RTL is a low level, triple based representation. The IR of CoSy [ACE00], CCMIR, is designed to represent imperative languages. OMIR extends CCMIR for object oriented languages as C++ and Java. CCMIR contains high level source language type information as Firm. For both, RTL and CCMIR/OMIR exists extensive documentation. The IR of lcc [FH01] is only documented as an interface between front end and back end. It is only designed for C and does not contain high level source type information. None of these IRs contain an inter-procedural representation.

The gcc compiler can build a SSA representation called GIMPLE on top of the tree IR. It adds Phi trees and establishes value numbering by introducing temporaries for each value.

.NET standardizes the intermediate binary format CIL [Lid] (common intermediate language). CIL is a stack language designed for target independent representation of a program. It serves as a common IR for various programming languages. It is not designed as a representation to perform optimizations on. It represents a program in triple form, and contains high level type information. It can be annotated with hints to conserve informations for optimizations.

0.2.2 IRs in Research

The SUIF2 [ADH$^+$] system is designed with similar targets as Firm. It supplies similar utility functionality, but no compiler specific algorithms or program optimizations. It offers a very flexible mechanism to manage compiler phases and to extend the IR. The predefined part of the program representation is close to an abstract syntax tree, i.e., it distinguishes statements and expressions.

The Scale [WMW96] compiler infrastructure is targeted to heterogeneous computing. Scribble [Scr], the IR of Scale, can represent programs written in C, Fortran and Java. It represents a program as a control flow graph, where the blocks are list of expression DAGs. Scale includes a phase to construct SSA form in Scribble by adding Phi DAGs in the expression lists and renaming variables. Scale performs basic optimizations comparable to those in Firm on this representation. It further implements a set of complex analyses and optimizations.

Neither SUIF nor Scale define a clear coupling between the type representation and the program code, complicating optimizations of polymorphic calls and the data layout.

The Trimaran [Tri] compiler infrastructure is designed as a research platform for compiling for new processor architectures. The infrastructure contains various high and low-level optimizations. It utilizes the IR Elcor [Elc] for most of its optimizations. Elcor is basically a control flow graph containing expression lists. The control flow graph contains special constructs to support back end specific algorithms as scheduling or generation of predicated instructions. It contains a data flow graph that represents dependencies between registers, but also dependencies through memory as does Firm. This graph is designed for back end algorithms, and therefore not well suited for high level analyses.

The Vortex [DDG$^+$96] compiler infrastructure is targeted to research on optimizations of object oriented languages. It includes a wide range of OO specific analyses and optimizations. The IR of Vortex represents high level type information comparable to Firm. The inheritance hierarchy is represented as a directed acyclic graph. Vortex does not represent the overwrites relation uniformly and explicitly as Firm does. Instead, it formulates fixed lookup rules combined with generic methods that group a set of polymorphic methods. Vortex also provides the possibility to fix the layout of complex data types in the front end or in a later phase. If the front end fixes the layout, high level information of field/method access or allocations is lost. Firm annotates the high level type description with the layout information, so that the high level information in the code representation is conserved. Optimizations violating the layout are forbidden. Vortex represents code as expression DAG lists in a control flow graph. It has no SSA representation nor standardized support for inter-procedural analyses as Firm. The compiler is implemented in the OO language Cecil.

0.3 Firm

Firm is the definition of an intermediate representation that evolved from the compiler research at University of Karlsruhe in the past 10 years. Its main merits comprise the explicit representation of program semantics as data dependencies in def-use-chains. Here it extends the ideas of [Cli95] to exceptions, the inter procedural data flow and others.

0.3.1 The Graph Structure

By definition Firm requires that a program is represented as a directed graph of elementary operations (jump, memory read/write, n-ary operation) such that each local variable is assigned exactly once in the program text. Actually, Firm completely resolves variables and only represents their values. Only references to such values may appear as operands in an operation. Thus, an operand explicitly indicates the data dependency to its point of origin. To extend this property past control flow Firm utilizes the SSA-form.

The directed Firm graph is an overlay of the control flow and the data flow graph of the program. Control operations branch to block operations. All non-block operations depend on a block operation. This makes them dependent on the control operations without introducing an artificial order.

SSA-form is a very elegant and easily comprehensible program representation as long as we only concentrate on handling local variables of the current procedure. Handling accesses to non-local variables, arrays, record fields, object attributes, etc., in general: handling of memory accesses, leads to additional complexities. [Tra01] introduced *explicit dependency graphs* (EDG), for dealing with these additional problems. On first sight, EDGs represent the memory as a huge, single atomic variable, the *state*, that is operand or result of operations depending on the memory. This introduces the necessary order to guarantee the correctness of the representation.

The data dependencies and the equal dependency on control operations through the block operation express the available ILP in the program. The refinement of the state variable allows to directly express certain alias and points-to information. With these concepts, Firm represents this information inherently and continuously over all compiler phases. Any analysis or optimization that contributes new insights adds these explicitly to the IR conserving them for later phases.

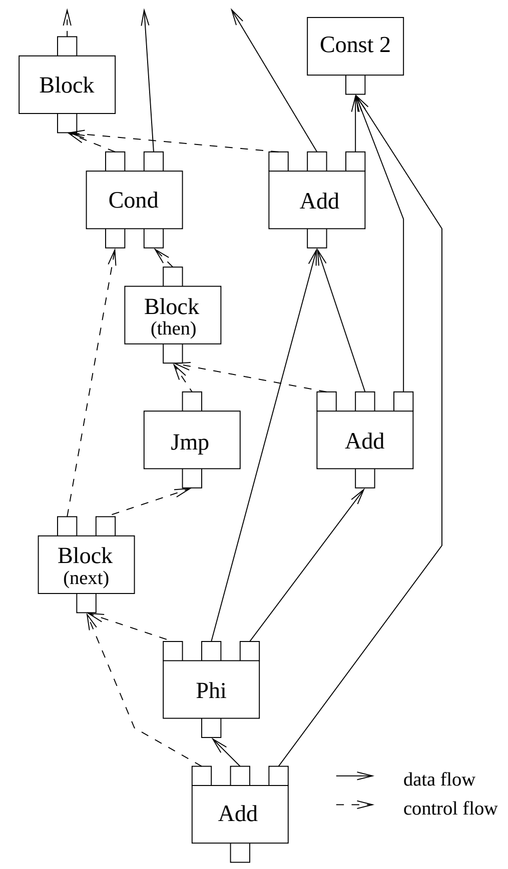
Firm graph edges are implemented as pure references to operations, directed backward. The IR can only be walked from the unique *End* operation of the graph. Operations computing unused results are not reachable from *End* and therefore fall out of the representation. In the IR described by [Cli95], the same happens to endless loops, as there are no control or data paths from an endless loop to the end of the procedure, resulting in wrong programs. Firm solves this by automatically detecting endless loops and storing the loop header block including important *Phi* operations as predecessors of the *End* operation.

As an example Fig. 1(ii) shows the Firm representation for the program fragment in Fig. 1(i). In the first basic block, the constant 2 is added to a. The conditional jump *Cond* passes control to the "then" or to the "next" *Block*, depending on the result of the comparison. In the "then" *Block*, the constant 2 is added to the result of the previous add operation. In the "next" *Block* two definitions of a reach the use of a in the addition. The *Phi* operation chooses which definition of variable a to use.

Figure 1 also shows that in the data flow representation the operations depending on the same *Block* operation are not sequentially ordered.

```
a := a + 2;
if ( ... ) {
   a := a + 2;
}
a := a + 2;
```

(i)



(ii)

Figure 1: The code in (i) is represented by the Firm graph in (ii).

0.3.2 Representation of Exceptions

Firm defines a novel representation of exceptions [Rie04]. This representation expresses the semantics of exceptions as control and data dependencies, integrating well with the concepts of Firm. The effects of exceptions are fully transparent to data flow analyses. The representation does not add additional basic blocks which would spoil the performance of data flow analyses.

Exceptions are side effects of certain operations. These side effects can alter the control flow. We call operations that can cause exceptions *fragile operations*. In Firm we model all side effects per definition as effects on the state variable. Fragile operations change this variable in some unspecified way if an exception occurs. Therefore Firm requires all fragile operations to have a state operand and a state result. This orders fragile operations with other operations with side effects sequentially.

To handle fragile operations in analyses without special casing for them, we represent their effects explicitly. Fragile operations have two control successors: the normal code and the exception handler. The exception handler either is a handler specified by the program source, or it indicates termination of a procedure.

In common IRs fragile operations end a basic block if they are defined as control flow operations. Fragile operations in Firm do not end basic blocks. Fragile operations have a single control flow result that passes control flow to a possible exception handler. If no exception occurs execution of the same block continues. I.e., with fragile operations basic blocks become extended basic blocks. This reduces the number of blocks in the representation. Other operations that do not depend on the fragile operation can easily be moved past this operation.

Depending on the construction of Firm we can express different restrictions on exception preciseness. To represent precise exceptions as required for Java, the exception handler evaluates the program state of the fragile operation in the program source. These exception semantics are comfortable for the user, but restrict optimizations considerably. Alternatively, Firm can represent the preciseness of exceptions on block or statement level. This allows to reset the context to the state before a "try" statement, leaving room for any optimization within this statement.

0.3.3 The High Level Type Representation

Firm tightly couples a type representation with the code representation. The semantics of several Firm operations depend on attributes referring to the type representation. This allows to represent compiler generated mechanisms as type casts and field and method selection on a high level basis.

The type representation differs only features useful for optimization and necessary for the translation. Complex type mechanisms must be resolved by the front end, but are represented explicitly in Firm.

Firm defines a generic representation of source language types. It distinguishes several kinds of types. *Primitive types* directly map to target machine modes. *Pointer types* directly map to the target pointer mode. In addition they specify the type of the instance they point to. *Method types* describe method interfaces. They list the number and type of arguments and results. *Array types* describe a collection of fields with the same type. These fields are called elements. An array type specifies a number of dimensions and an element type. Optionally it specifies the size of the dimensions. *Union types* describe several interpretations of the same memory region. *Struct types* describe a collection of fields with different types.

Class types describe a partial collection of fields and methods. They list a set of super types. The collection of fields and methods is completed by inheriting fields and methods from the super types. Fields and methods of class types specify polymorphy by referencing the fields/methods in super-classes they overwrite. If a field is overwritten in a subclass, it is not inherited to this class. Firm allows to explicitly resolve the inheritance relation. It knows fields and

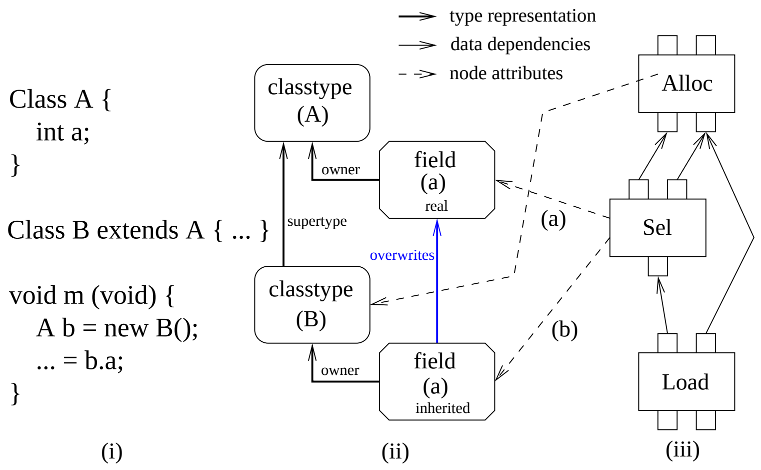


Figure 2: Restriction of the program representation with inherited fields. (ii) shows explicit representation of inheritance of the code in (i). The field *a* of class type *B* was added by the compiler to explicitly resolve the inheritance. (iii) shows restriction of the *Sel* operation: Replacing edge (a) by (b) expresses explicitly, that *Sel* never accesses the field a in an instance that is of type A but not of type B.

methods that are marked as *inherited*. Such fields or methods must specify that they overwrite the field or method they originate from. See Fig. 2, (i) and (ii) for an example.

Relying on this type representation, Firm defines the following high level operations to support analyses.

The *Sel* operation models feature access in a single operation. It fully abstracts from pointer arithmetics, dispatch mechanisms or other functionality utilized to implement the accesses. *Sel* has as operand an address of an instance of a complex type. An attribute of the operation specifies which field or method it selects from this instance. The result of the operation is the address of this field or method.

An analysis must look at a single operation to interpret a field access. If the analysis determines that a polymorphic method selection only selects a method of a certain subclass it can express this information explicitly in the IR by setting the corresponding attribute in the *Sel* operation. Fig. 2(iii) shows an example.

The source type *Cast* operation of Firm abstracts explicit type casts as necessary for multiple inheritance. All sources of values, as *Load* and *Const*, point to the type of the value they represent. The *Alloc* operation knows the type of the allocated data structure. This allows to fully type the IR. Analyses can use the *Cast* operation to express information about the type of a value they determined. Interpreting the *Cast* operation reduces the values on a data flow edge. This is especially useful for pointer values.

Firm does not fix the layout of complex types, i.e., optimizations can alter the layout. A lowering phase can fix the layout of complex data structures and introduce dispatch tables in the type representation. Then it can lower the high level IR operations by introducing method dispatch and cast functionality. This lowering phase can take advantage of knowledge about the compiled source language, as, whether the language allows multiple inheritance or not. Naturally, one can also construct Firm using only low level constructs, if the source language requires this.

0.3.4 Explicit Dependency Graphs

Heap analyses profit from a SSA like representation of variables in memory. Nevertheless we do not want to represent each variable in single assignment form, as this means that we have to generate a copy of a variable at each assignment to the variable. It is unlikely that a deconstruction of a representation that generates copies results in a program with comparable memory consumption as the original program. Further representing these copies means large memory consumption for the representation of the analysis values. Therefore a functional representation of heap variables must retain the memory locations.

[Cli95] proposes to represent the memory as a value, the state, that is a result of operations that can change the memory (as *Store* and *Call*). It is an operand to all operations that access the memory. These dependencies represent Def-Use dependencies between non local variables. [Cli95] does not represent Use-Def dependencies. He assures program correctness by placing all loads right after the operation they depend on by definition.

[Ste95] extends the representation of [Cli95] to a representation of partial states. These allow to describe the Def-Use dependencies depending on the variables potentially accessed by the operations. This representation unveils more parallelism in the program and increases the effectiveness of analyses.

Both representations can not express past which succeeding memory defining operation a load can be moved. Explicit dependency graphs [Tra01] (EDGs) extend these state representations by adding explicit Use-Def dependencies between operations. Especially the *Load* operation has the state value as result. With this representation only operations that depend on the same part of the state depend on each other. This automatically allows to represent partial states. These states are represented in SSA Form with special PhiS operations. The EDG operation *Sync* can unite several partial state values if they can no more be distinguished. Fig. 3 shows the advantage of Firm.

States, and especially partial states, speed up analyses evaluating the content of non local variables. To compute the result of a load operation, the analysis must consider all operations that are a reaching definition. Walking the state edges in the Firm graph allows to reach relevant operations without visiting any other. Introducing more partial states allows to express information gathered with an alias or points to analysis directly in the IR. Without additional analyses all memory accesses must be held in strict temporal order; reordering may lead to wrong values being fetched from memory.

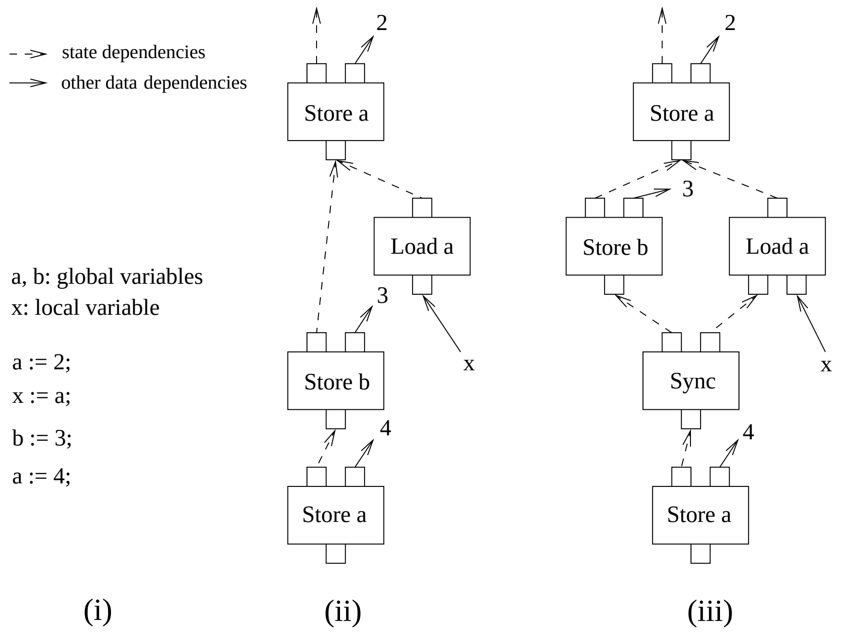


Figure 3: The Firm extension to the representation of states. In (ii) the *Load* depending on the first *Store* must implicitly be considered executed before the second *Store*. (iii) shows how Firm expresses that the *Load* and the second *Store* can be executed in parallel.

Additionally, EDGs extend the IR to an inter-procedural data flow graph. This graph directly connects all call sites with called methods. *Call* operations branch to the possibly called methods as determined by a call graph analysis. The graph links the actual parameters at call sites to the formal parameters in the called procedures. We merge parameters from several call sites with *Phi* operations. The method return is represented accordingly. By this the inter- and intra-procedural representations are uniform. A data flow analysis can operate on an intra procedural graph and on an inter-procedural graph without major changes.

0.3.5 Firm in a compiler

A Firm representation may easily be generated during a tree walk through the attributed syntax tree as generated from a compiler front-end. The interface to the front end integrates an algorithm that constructs the SSA-form and the initial state dependencies automatically.

Firm applies a set of optimizations to an operation right after its construction. Applying a set of optimizations to a single operation before applying the same optimizations to its users unveils better optimization possibilities than applying each optimization to the whole program. Our algorithm guarantees that unnecessary operations are not allocated at all. This keeps the program representation small from the very first.

Firm unites a high and low level IR. A compiler must not use an other representation between semantic analyses and code generation. Therefore we assume the code generation phase directly deals with SSA deconstruction.

0.4 Applications and Implementation of Firm

We utilize Firm in various contexts. Its initial design is derived from our experiences with the IRs LABIL and BABIL, and the work in [Cli95]. [AvR96] implemented the first version of Firm in the Sather-K compiler fiasco. [Tra01] refines Firm and proposes a heap analysis on Firm. The work of [Tra01] is also integrated in the fiasco compiler.

[TLB99] describes the first stable version of Firm. We implemented this version in the stand alone library libFirm [Lin02]. The AJACS project [Gau02] uses libFirm in a Java compiler for embedded automotive applications. Jack, a Java compiler based on the jikes front end, and CRS, a C compiler based on the gcc front end, both use libFirm as IR. Finally we use libFirm along with Eli [KPJ98] in the compiler construction laboratory at University of Karlsruhe for now 5 years to teach the handling of SSA and optimizations, which allowed us to improve interface and documentation based on user feedback.

The EU research project Joses [GAF$^+$99] defines Firm in fSDL [Buh93, WKD95], a specification language for IRs in the commercial CoSy compiler framework [AAvS94] distributed by ACE. The project utilizes a Firm view [Lin00] on the CoSy IR CCMIR for optimizations as inlining of Java array objects.

The DfG research project CATE [cat] investigates the use of data flow analyses in a meta programming system. In CATE we integrated libFirm [Kle03] in the meta programming tool Recoder [LH00].

0.4.1 Firm implementation

Firm represents all entities (as types, stack frames, global variables) with the same mechanisms reducing the learning efforts. It defines only 36 basic operations with fixed semantics for the program code representation. Implementations of new algorithms must only deal with this small number of operations.

Firm utilizes type tags for run time type checking to simplify developments on top of it. Checker routines verify the representation to find invalid constructs introduced by transformations. The interface to Firm uses data encapsulation to reduce the effects of changing the internal implementation on external phases.

We implemented a variety of standard optimizations on Firm including:

- constant evaluation, algebraic simplification and reassociation
- unreachable code and dead code elimination
- control flow optimizations (straightening, weak and strong if-simplification, removal of critical control flow edges)
- common subexpression elimination, code placement, partial redundancy elimination and strength reduction
- inlining and tail recursion

As well as the following analyses:

- construction of SSA form (integrated in the interface to the front end)
- reversion of the directed data flow graph (Def-Use edges)
- dominator information
- back edges and strongly connected regions
- several loop trees for data flow work list algorithm
- basic call graph analysis
- rapid type analysis [BS96]

Firm is implemented in C but includes a Java Native Interface, so that it can be used with front ends programmed in Java.

0.5 Experiments

In this Section we compare the complexity of Firm with the complexity of the SSA representation GIMPLE [Mer03] implemented in the gcc compiler (version `3.5-tree-ssa`). We measure the number of operations needed to represent a program in each of the IRs. We use CRS to construct Firm. CRS uses the gcc front end version `3.3.2`. We chose GIMPLE for a comparison as it is constructed from the same front end as Firm, ruling out spurious effects.

The number of operations is significant for compiler performance. Less operations means faster traversals over the IR and less memory consumption for the program representation. Comparing the runtime of compilers or optimization phases is misleading. This merely compares the performance of the optimization, not of the IR. Comparing the runtime of resulting executables compares the quality of the optimization result, but not the quality of the IR. An optimization written with less engineering effort and running faster must not produce better optimization results.

We count the operations needed in the Firm representation right after the construction is completed. We apply the basic optimizations during the construction as proposed by [Cli95]. This is a basic advantage of Firm over other IRs. On other IRs these optimizations are hard, as they require a data flow analysis. Further it keeps the representation small from the very beginning. The representation never contained more operations than we count. Moreover we can apply the optimizations without an own pass over the IR.

We measure the gcc SSA representation right after its construction in pass "rewrite_into_ssa" by iterating over the control flow graph and using gcc's own walker function for the tree expressions. Gcc executed a set of optimizations as constant folding during the construction of its non-SSA IR.

For both representations, we count the Phi operations separately. In Firm, we additionally count the PhiS operations (see Sec. 0.3.4). We do not count operations needed to represent types or global variables.

With both compilers we translate all C programs in the SPEC00 benchmarks. Table 1 shows the results of the experiment. Fig. 4 compares the numbers.

Table 1 lists in colums labled "ops" the overall number of all operations including the Phi and PhiS operations in thousands. The columns labeled "Phi" list the number of Phi nodes. For Firm we separatley list the number of PhiS operations.

Fig. 4 shows bars for operations (light gray) and Phi nodes (dark gray). The bars represent the percentage of nodes in the Firm relatively to the nodes in GIMPLE. The percentage for Phi operations does not consider the PhiS operations as these represent a feature not available in the GIMPLE representation.

The numbers show that the representation of a program in Firm needs only 47% of operations compared to GIMPLE. This is a considerable reduction. Firm requires even less Phi operations: only 20%. The representation of the heap as an explicit variable in Firm induces a considerable number of extra Phi operations (column PhiS).

Table 1: Number of operations (ops in thousands) and Phi nodes

	Firm			GIMPLE	
Program	ops	Phi	PhiS	ops	Phi
164.gzip	13	532	543	30	3093
175.vpr	42	1096	1360	98	3348
176.gcc	571	15350	22431	1077	87557
177.mesa	153	4299	4139	322	4478
179.art	3	184	153	9	397
181.mcf	3	134	143	6	130
183.equake	5	116	160	17	763
186.crafty	56	1383	2268	228	36179
188.ammp	38	1269	1021	75	1464
197.parser	32	1119	1366	64	3735
253.perlbmk	219	6777	9294	445	35920
254.gap	200	7564	8074	438	14700
255.vortex	160	2557	8804	353	26077
256.bzip2	9	372	391	18	1333
300.twolf	80	2948	2920	201	9152
Sum	1591	45700	63067	3389	228326

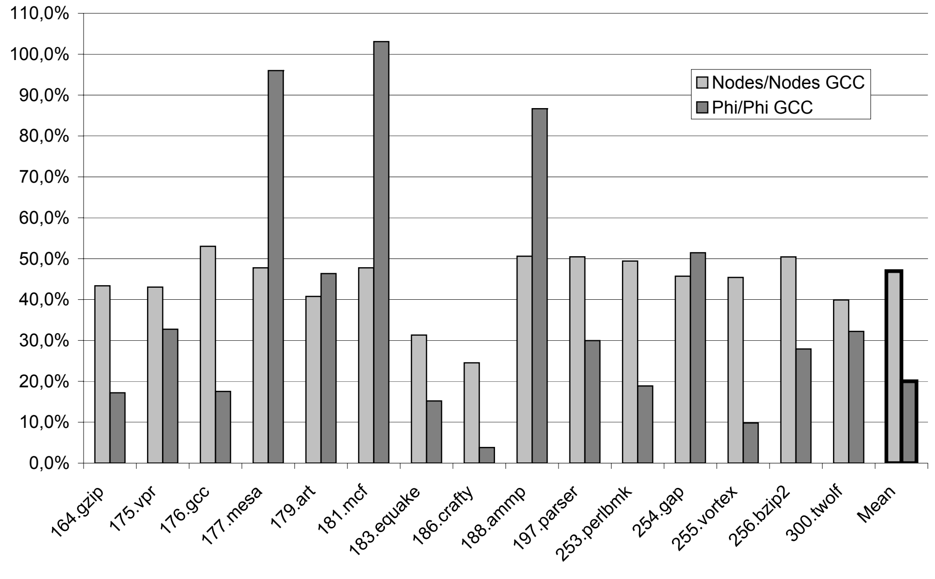


Figure 4: The code in (i) is represented by the Firm graph in (ii).

Mesa, mcf and ammp have significant more Phi operations in Firm than the other programs. Gcc 3.5 merges the conditions of directly nested if statements on the AST and constructs simplified control flow. This represents the same semantics with different control flow requiring less Phi operations. On Firm we apply this optimization only later, when we recognize empty blocks.

Crafty and vortex have an extreme low number of Phi operations in Firm. The chess simulator crafty uses a lot of global constants to express positions on the field an the like. Firm evaluates these better, removing dead code and avoiding Phis. The differences in vortex are in functions that use a lot of local variables in inner blocks of deeply nested loops. Only very few definitions of these variables are reachable in other blocks. In both programs GIMPLE requires many Phi operations, the fraction of Phis of all operations is relatively high, over 10% in crafty, indicating that GIMPLE can not detect that these values are dead. The

0.6 Conclusion

In compiler construction the IR is the central representation for program analysis and optimization. We introduced our IR Firm, a graph-based SSA-representation with well defined semantics. It does not rely on a specific source language but can represent programs in different types of languages: object-oriented and imperative languages, like C or Java. It preserves the high level abstraction and does represent OO features.

Due to the careful design of the operations and the superior structure, Firm expresses the semantics of a program concise, using few operations. The construction of Firm graphs is efficient. Already during this construction phase standard optimizations like constant folding and common sub-expression elimination are performed, which keeps the graphs even smaller. In comparison to the gcc SSA-representation GIMPLE we represent programs in equivalent Firm graphs with 53% less operations and 80% less Phi operations.

We introduce an exception representation that integrates well with the common representations of control and data flow. This has the benefit that operations, which can raise an exception, must not be handled specially. Firm is a platform for various analyses and optimizations. Heap analyses are supported by the novel sparse functional representation of the heap. Firm can represent the results of analyses directly in the IR. Furthermore, Firm offers an uniform view on graphs for inter- and intra-procedural analyses.

Firm has been used in practical courses by students, in research and industrial projects as well.

In our current work we investigate new, and implement known analyses and optimizations on Firm. Especially, we focus on heap analysis and the optimization of object-oriented programs. We explore the benefits of data flow analyses in the area of meta programming. Ongoing research in back ends considers integrated code selection and SSA deconstruction resulting again in a graph based representation. Register allocation and scheduling then directly consume this graph.

Bibliography

[AAvS94] Martin Alt, Uwe Aßmann, and Hans van Someren. Cosy compiler phase embedding with the CoSy compiler model. In *Proceedings of the 5th International Conference on Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 278–293. Springer-Verlag, April 1994.

[ACE00] ACE Associated Compiler Experts. CCMIR Definition – Specification in fSDL, Description and Rationale. Technical report, ACE Associated Compiler Experts bv, Amsterdam, The Netherlands, 2000.

[ADH$^+$] Gerald Aigner, Amer Diwan, David L. Heine, Monica S. Lam, David L. Moore, Brian R. Murphy, and Constantine Saputzakis. An Overview of the SUIF2 Compiler Infrastructure. Technical report, Computer System Laboratory, Stanford University, http://suif.stanford.edu/suif/suif2/doc-2.2.0-4/overview.ps.

[AvR96] Markus Armbruster and Christian von Roques. Entwurf und Realisierung eines Sather-K Übersetzers. Diplomarbeit, Dept. of Computer Science, University of Karlsruhe (TH), December 1996.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 1–11, San Diego, CA, USA, 1988. ACM Press.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *1996 ACM conference on Object–Oriented Programming Systems, Languages, and Applications*, volume 31, pages 324–341, San Jose, CA, USA, October 1996. ACM Press.

[Buh93] Claus-Thomas Buhl. Ein Übersetzer für das CoSy-Modell. Diplomarbeit, Dept. of Computer Science, University of Karlsruhe (TH), July 1993.

[cat] CATE – Component Analysis and Transformation Engine. Website, Dept. of Computer Science, University of Karlsruhe (TH), http://www.info.uni-karlsruhe.de/projects.php/id=54.

[Cli95] Clifford Noel Click Jr. *Combining Analysis, Combining Optimizations.* PhD thesis, Dept. of Computer Science, Rice University, February 1995.

[DDG+96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *1996 ACM conference on Object–Oriented Programming Systems, Languages, and Applications*, volume 31, San Jose, CA, USA, October 1996. ACM Press.

[Elc] The Elcor Intermediate Representation. Documentation, Compiler and Architecture Research Group, Hewlett Packard Laboratories; IMPACT Group, University of Illinois; Center for Research on Embedded Systems and Technology (CREST), Georgia Institute of Technology, http://www.trimaran.org/docs/elcor_ir_manual.ps.

[FH01] Christopher W. Fraser and David R. Hanson. The lcc 4.x Code-Generation Interface. Technical Report MSR-TR-2001-64, Microsoft Research, July 2001.

[GAF+99] Daniela Genius, Uwe Aßmann, Peter Fritzson, Henk Sips, Rob Kurver, Reinhard Wilhelm, Henk Schepers, and Tom Rindborg. Java and CoSy Technology for Embedded Systems: the JOSES project. In J.-Y. Roger et. al., editor, *Proc. of the European Multimedia, Microprocessor Systems, Technologies for Business Processing and Electronic Commerce Conference (EMMSEC'99)*, Amsterdam, June 1999. IOS Press.

[Gau02] Thilo Gaul. AJACS: Applying Java to Automotive Control Systems. *Automotive Engineering Partners*, 4, August 2002.

[Kle03] Olaf Kleine. Erweiterung eines Metaprogrammiersystems um einen iterativ aktualisierbaren Zwischensprachaufbau. Diplomarbeit, Dept. of Computer Science, University of Karlsruhe (TH), May 2003.

[KPJ98] Uwe Kastens, Peter Pfahler, and Matthias Jung. The Eli System. In Kai Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction*, volume 1318 of *Lecture Notes in Computer Science*, pages 294–297. Springer-Verlag, March 1998.

[LH00] Andreas Ludwig and Dirk Heuzeroth. Metaprogramming in the Large. In *Net.ObjectDays 2000 Tagungsband, 2nd International Conference on Generative and Component-Based Software-Engineering*, pages 443–452, October 2000.

[Lid] S. Lidin. *Inside Microsoft .NET IL Assembler.* Microsoft Press.

[Lin00] Götz Lindenmaier. Design of Conflict and Access Analysis. Technical Report 8502, The JOSES Consortium, May 2000.

[Lin02] Götz Lindenmaier. libFIRM – A Library for Compiler Optimization Research Implementing FIRM. Interner Bericht 2002-5, Dept. of Computer Science, University of Karlsruhe (TH), September 2002.

[Mer03] Jason Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. In *Proceedings of the GCC Developers Summit*, pages 171–193, May 2003.

[Rie04] Till Riedel. Ausnahmebehandlung in einem Optimierenden Javaübersetzer. Studienarbeit, Dept. of Computer Science, University of Karlsruhe (TH), June 2004.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 12–27, New York, NY, USA, January 1988. ACM Press.

[Scr] Scribble Control Flow Graph. Documentation, Dept. of Computer Science, University of Massachusetts, Amherst, http://www-ali.cs.umass.edu/Scale/cfg.html.

[Sta02] Richard M. Stallman. GNU Compiler Collection Internals. Technical report, Free Software Foundation, http://gcc.gnu.org/onlinedocs/gccint.ps.gz, December 2002.

[Ste95] Bjarne Steensgaard. Sparse functional stores for imperative programs. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*, volume 30 of *ACM SIGPLAN Notices*, pages 62–70, March 1995.

[TLB99] Martin Trapp, Götz Lindenmaier, and Boris Boesler. Documentation of the Intermediate Representation FIRM. Interner Bericht 1999-14, Dept. of Computer Science, University of Karlsruhe (TH), December 1999.

[Tra01] Martin Trapp. *Optimierung Objektorientierter Programme. Übersetzungstechniken, Analysen und Transformationen*. PhD thesis, Dept. of Computer Science, University of Karlsruhe (TH), October 2001.

[Tri] Trimaran – An Infrastructure for Research in Instruction-Level Parallelism. Documentation, Compiler and Architecture Research Group, Hewlett Packard Laboratories; IMPACT Group, University of Illinois; Center for Research on Embedded Systems and Technology (CREST), Georgia Institute of Technology, http://www.trimaran.org/index.shtml.

[WKD95] H. R. Walters, J. F. Th. Kamperman, and T. B. Dinesh. An extensible language for the generation of parallel data manipulation and control packages. Technical report, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1995.

[WMW96] G. Weaver, Kathryn S. McKinley, and C. Weems. Score: A compiler representation for heterogeneous systems. In *Heterogeneous Computing Workshop*, pages 10–23, Honolulu, HI, USA, April 1996.