Authors:
Andrea G. M. Cilio (A.Cilio@its.tudelft.nl)
Henk Corporaal (H.Corporaal@its.tudelft.nl)

corresponding author and presenting author:
Andrea G. M. Cilio (A.Cilio@its.tudelft.nl)

A Linker for Effective Whole-program Optimizations

Andrea G. M. Cilio and Henk Corporaal

Delft University of Technology
Computer Architecture and Digital Techniques Dept.
Mekelweg 4, 2628CD Delft, The Netherlands
A.Cilio@its.tudelft.nl H.Corporaal@its.tudelft.nl

Abstract. The use of a standard binary format in the later part of code generation promotes efficiency and interchangeability of tools, but leaves little information on the source file in the machine code representation. We propose a new approach to code generation, based on a single, highly structured internal format used during proper compilation, machine code generation and linkage. This format offers new opportunities for *whole-program* optimizations. We have implemented and tested a code generator based on this format. Although the use of traditional a binary format is more efficient, we believe that the increase in code size and compilation times are largely compensated by the opportunities offered by this new trajectory. To support this assertion, we discuss some of its potential applications.

1 Introduction

A traditional compilation path is split into three phases. The first phase, the actual compilation, transforms the source program into assembly code. To support the many optimizing code transformations performed during compilation, a rich internal intermediate representation, which usually preserves most of the source-level information, is used. The intermediate representation is then translated into code for the target architecture, and a text assembly file is written out. During this step, a large part of the source-level information is lost. In the second phase, the assembler parses the assembly file and translates the code into the binary format of the target machine. During this process, an offset relative to a program section is assigned to every variable and machine instruction, and relocation information is built up. The resulting output file is called *object file*. The third step is the linkage of one or more object files and libraries into an executable binary file. During linkage, program sections from different modules are merged into a single file and symbolic references to data objects or instructions are *resolved*, i.e., replaced with the actual physical address of the referenced symbol.

This well-established code generation path offers three advantages for general purpose target processors. First, it is *efficient*: the binary format is a compact representation; linking modules and libraries does not require expensive code transformations. Second, it becomes possible to distribute library code without the source files. Third, since assembly syntax and binary format are standardized with respect to target architecture and operating system, it promotes *interchangeability* of the tools that perform assembly and linkage, provided that they comply to these standard formats.

This compilation approach is also common for Application Specific Instruction-set Processors (ASIPs). Code generators are often derived by re-targeting tools originally designed for general purpose processors [9]. This approach, although popular, has some drawbacks. Interchangeability of binary tools is less relevant, because these are usually ad-hoc designs for a single (possibly *templated*) architecture. Moreover, the concern is now shifted towards the "quality" of the code produced rather than the speed of code generation. At the same time, a number of new requirements become critical when generating code for ASIPs and, more generally, for instruction-level parallel processors. Altogether, these requirements force us to completely change the compilation trajectory.

The rest of this paper is organized as follows. Section 2 describes the new requirements of a code generator for ASIPs and the guidelines for its design. In Sect.3 we present our implementation of the proposed compilation path, while Sect.4 describes the linker in detail. Section 5 summarizes some of the opportunities offered by this new trajectory. In Sect.6 we review related work. Finally, Sect.7 concludes this paper.

2 Requirements and Design of a New Compilation Trajectory

This section details the requirements of a code generation path geared towards ASIPs. From these requirements we derive the guidelines for the design of a new compilation path. In short, the semantic level of object file representation has to be risen. The intermediate format used in the first compilation phase has to integrated with the format used to represent the program at the machine-code level.

Requirements of code generation for ASIPs. To effectively exploit the resources offered by a modern architecture, a compiler must apply all known state-of-the-art optimizations. This need is even more critical when compiling for ASIPs, as every code improvement not only reduces the execution time, it may also contribute to reducing the overall cost of the system being designed.

Inter-procedural optimizations are normally restricted to functions in the same compilation unit. No information is available on functions that have external linkage during compilation. When it is guaranteed that no more code is to be included, the optimizer is not forced to make conservative assumptions about code not yet linked and applying optimizations on *whole programs* has great potential for improving the code. Since whole-program optimizations require that all modules be linked together, they are best performed during or after linkage. However, the binary formats commonly used at link-time are inconvenient for complex code transformations. The new format, destined to supersede the binary format, must thus allow to easily perform analyses and transformations on the machine code after assembly and linkage.

To reduce the design cost, ASIPs are often built according to a template. A target processor is obtained by instantiating the architectural parameters of the template, and that the code generator must address a possibly huge space of target processors. How does linkage of libraries fit into this scheme? Clearly, generating library code for every possible target configuration in advance is impossible. Linkage of source files (referred to as *monolithic compilation*) or at the machine-independent intermediate level has also a severe drawback. Compilation of large files is highly expensive and can unacceptably stretch the design cycle. A short design cycle is essential for automatic exploration of the design space of an ASIP architecture. In this context, finding a cost-effective

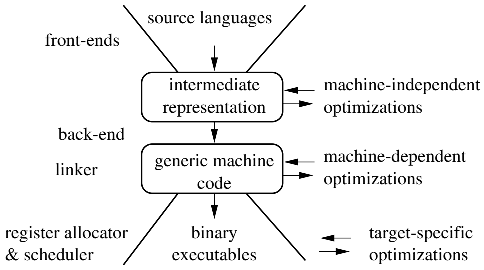

Fig. 1. Code generation for a templated target architecture; the application can be coded in one of the supported source languages and compiled to a binary for one of the many possible targets.

solution may require to compile for hundreds of different target configurations [2]. A good solution is to generate object code and libraries at the level of a *generic* machine. Generic machine code should capture the aspects of the templated architecture common to every instantiation, like the base instruction set. Aspects dependent on the processor instantiation, like hardware resources and parallelism available, are dealt with subsequently, in a separate scheduling and allocation phase. The generic machine code is an ideal level of abstraction for object files and linkage, because it takes the best of the two previous strategies: it limits the amount of target-specific passes to perform after linkage while keeping the number of pre-compiled libraries at a minimum. Figure 1 tries to give a conceptual view of this scenario.

Guidelines for a new code generation path. The requirements exposed above naturally lead to the organization of a new format representing machine code. To facilitate symbolic analysis and code manipulation, the machine code must be represented in symbolic form. E.g., while the binary representation is bound to the *memory layout* of the program, in our representation the memory layout is just an additional piece of information. References to data and code are pointers to symbol table entries. As a result, a code optimization pass needs not keep track of offset changes to addresses and branch displacements during the code transformation. The scope hierarchy is represented by means of a tree of symbol tables. A crucial piece of information for some optimizations, like memory disambiguation of pointer-based data structures [6], is type information. The intermediate format must be capable of representing the type system of the source language. An effective way to preserve high-level information down to the machine level, is to derive the new format from the machine-independent intermediate format. The semantics of the target instruction set and the architectural registers are added to the new format, while its overall structure is left unmodified.

3 The Compilation Trajectory

This section presents the retargetable code generator that implements our new approach. As stated in Sect.2, we generate code for an ample spectrum of target processors sharing the same templated architecture. This architecture offers explicitly programmed

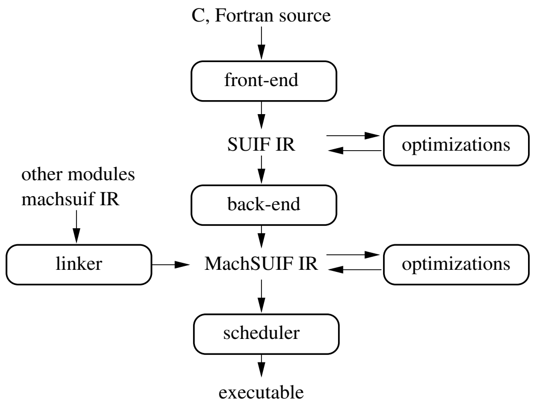

Fig. 2. Scheme of the proposed code generation trajectory.

instruction-level parallelism, in a fashion similar to that of VLIW architectures [2]. The code generation is coarsely split in two phases: (1) compilation to the generic-machine instruction set and (2) target-specific, integrated scheduling and register allocation [8]. The first phase will be the focus of this paper.

The SUIF representation and the front-end. Our code generator is based on the SUIF compilation system [5] and its intermediate representation, the Stanford University Intermediate Format (SUIF). Every pass of the fist phase is implemented as a SUIF-to-SUIF transformation. The SUIF format can represent the source program semantics at different, machine-independent levels of abstractions. Figure 2 gives an overall view of our code generation trajectory. SUIF instructions capture source-level information like the array indexing, multi-branch statements, and the types used in the program. At this level, several analyses and optimizations that require source-level information, like loop restructuring and memory disambiguation, are performed.

Code generation and section offset assignment. The compiler back-end maps SUIF instructions into generic machine code. The machine instructions are represented in *MachSUIF*, a format that extends the SUIF representation for machine-specific compilation and code optimizations [10]. Differently from most compilers, the translation to machine code fully preserves the high-level information present: only the instruction representation is extended, to encapsulate machine-specifics like opcodes, registers and addressing modes.

In our compilation path, the typical assembly step is bypassed, in that the machine instructions are not translated into their actual binary encoding yet. Symbols are tagged with their program section and the offset within this section by means of a *SUIF annotation*, called "section_offset", shown in Fig.3. The section ID number uniquely defines the section. As only three IDs are currently used, it is possible to define additional sections for special purposes. The offsets are expressed in the addressable unit relative to their section; e.g., instruction words for the TEXT section. The symbols that have an

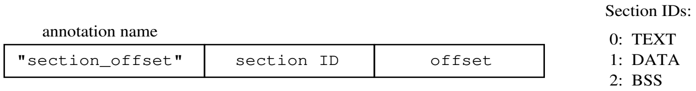

Fig. 3. The format of a "section_offset" SUIF annotation.

offset are: global and *private* variables, procedure and label symbols. By private symbol we mean a global symbol that visible only in the source file where it is defined.

This organization allows to conveniently lookup a symbolic reference and its section offset in a given symbol table. Offset assignment is implemented by iterating over all symbols defined in a symbol table and recursing over the tree of symbol tables. The resulting MachSUIF output file corresponds to a binary object file of the traditional trajectory. This object file can then be linked with other library or object modules, by our newly designed MachSUIF linker. An important benefit of using SUIF annotations to record section offsets is that it is *transparent* to other MachSUIF passes. As a result, code optimization passes can be reordered before or after section assignment or linkage.

4 Linking MachSUIF Files

This section describes the implementation of the linker. Apart from the alignment restrictions and the word sizes, the approach presented does not depend on the target architecture. Since alignments and word sizes are encapsulated under a uniform interface in MachSUIF, any trajectory based on MachSUIF can use this linker, provided that the section offsets are computed as described in Sect.3.

4.1 Basic approach for linking MachSUIF files

The MachSUIF linker performs two functions. First, it merges symbol tables coming from different modules and, for each program section, merges the offsets into a single offset space. Second, it resolves references to external, library symbols.

Section offset merging is easily implemented. Every module occupies a segment of the section offset space starting from zero; the linker simply adds to the offset of every symbol the base offset relative to its section. The base offsets are the section sizes resulting from summing the offset space taken by the files linked before the current one. This means that the offset spaces of each module are shifted by a constant amount.

Unresolved references to an external symbol s are resolved by function $\mathrm{Resolve}()$ as follows. The name of s is looked up in a given library symbol table. If the symbol, s', is found, all the references (i.e., pointers) to s' are replaced with references to s. This guarantees that any library function or variable that will be linked references the symbol in the output file. If s is a variable, a unique section offset and the initializer of s', are assigned to s.[1]

[1] For procedures, this process is postponed.

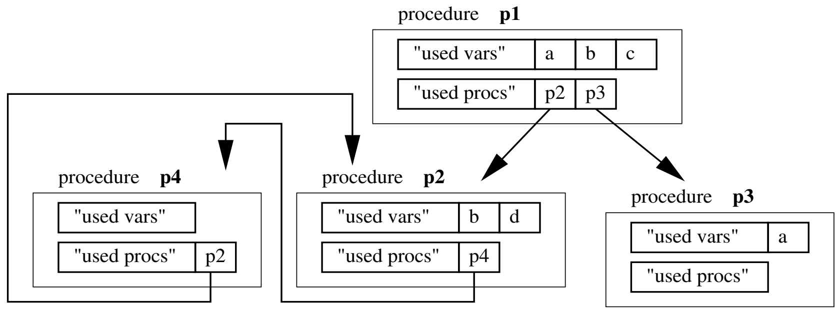

Fig. 4. Reference graph of library procedures.

When linking a library file, only the symbols which are referenced in one of the object files have to be linked to the output file. Therefore, the simple offset computation described for object files is not applicable. A MachSUIF library file differs from normal MachSUIF object files. First of all, the offset of global and private symbols is stripped off, as there is no need to maintain a relative order between symbols' offset. Second, the segments of offset space relative to each procedure are 'relocated' to base offset zero. The motivation for this is that we want to maintain the relative offset order between procedure's symbols, and avoid to recompute the alignments.

An external symbol that is resolved in a library may contain references to other library symbols which need to be resolved. This process must iterate until no new unresolved references are added. Note that some of these references may be to private symbols of the library. Detecting symbol references within SUIF procedures is time-consuming, as it requires scanning all symbol table entries and MachSUIF instructions of the procedure. Therefore, we perform it once and for every procedure when we *build* the library MachSUIF file. For each procedure we record all the non-local symbol references found in two *reference lists*; one list contains references to variables, one references to procedures. The lists are attached to the procedure's symbol table by means of two annotations, called "used_vars" and "used_procs", respectively. This organization defines a directed graph of procedure references, as shown in Fig.4. Starting from a procedure, all references to the library symbols that need to be resolved can be captured by traversing the graph. Note that, as opposed to the call graph, this reference graph includes procedures that are not called, but whose addresses are used.

4.2 Algorithm for resolving references to library symbols

Figure 5 shows the algorithm used to resolve global symbols defined by a library. First, the symbol table of the output file is scanned. Every external symbol defined in the library symbol table is resolved. A resolved variable v may contain symbolic references in its initializer, $\mathrm{VarDef}(v)$. A resolved procedure p has its symbol references recorded in the reference list. The function $\mathrm{FindRef}()$ retrieves all the unresolved references of the variable initializer, while the function $\mathrm{References}(p)$ selects the symbols in p's reference lists as-yet to be resolved. Unresolved references are recorded in the temporary

```
procedure ResolveLibraryReferences(library_symtab, out_symtable)
beginproc
    reflist := ∅
    foreach extern variable v in out_symtab do            /* resolve external variables */
        if v is defined in library_symtab then
            Resolve(library_symtab, v)
            reflist := reflist ∪ FindRef(VarDef(v))
        endif
    endfor
    foreach extern procedure p in out_symtab do           /* resolve external procedures */
        if p is defined in library_symtab then
            Resolve(library_symtab, p)
            reflist := reflist ∪ References(p)
        endif
    endfor
    while reflist ≠ ∅                         /* resolve indirectly referenced library symbols */
        foreach global library variable v′ ∈ reflist do
            Resolve(library_symtab, v′)
            reflist := reflist − v′
        endfor
        foreach library procedure p′ ∈ reflist do
            if p′ has external linkage Resolve(library_symtab, p′)
            reflist := (reflist − p′) ∪ References(p′)
        endfor
    endwhile
endproc
```

Fig. 5. Algorithm for resolving all direct and indirect references to global library symbols.

list *reflist*. After all external symbols defined in the library are resolved, *reflist* contains all the library symbols that still have to be linked to the output file. First, every global variable v' is resolved[2] and removed from *reflist*. This time, we do not need to search for possible references in $\mathrm{VarDef}(v')$, because such references have been recorded in the reference lists of the same procedure that referenced v'. Then, every procedure p' is resolved and removed from *reflist*, while its unresolved references are added to *reflist*. This process is iterated until *reflist* is empty.

5 Possible Applications

This section motivates our approach by presenting a number of applications taken from the literature that benefit from the opportunities offered by the new trajectory. We also describe a novel application that we intend to implement. This application is made possible by the combination of source-level type information and the possibility to conveniently analyze and modify linked code.

[2] Private symbols are resolved in a later pass.

Inter-module optimizations. By using the MachSUIF format for object files and linkage, all optimizations requiring inter-procedural analysis can be easily extended across compilation units. One such optimization is *static allocation* of local variables [9]. A variable is statically allocated if its location is known at compile time. Local variables of different procedures can share the same location if it is guaranteed that these procedures belong to disjoint branches of the call graph. This means that every possible call path leading to a function execution must known. When applicable, static allocation is preferable to stack allocation, because it eliminates address computations and instructions to save and restore the stack pointer. Note that in a traditional compiler static allocation is restricted to private functions, because the calls to a function with external linkage in other source files are not known at compile time. Another important optimization is inlining of external functions. When performed at machine level, function inlining requires redoing register allocation and scheduling of the inlined code. Our trajectory avoids this drawback, because inlining is performed on the generic machine code before scheduling and register allocation.[3]

Optimizations using high-level information. A number of low-level transformations, particularly those increasing the instruction-level parallelism of the program, are more effective when the information computed by source-level analyses is available [7]. Our proposal is to compute the required information at the point in compilation which is most suitable for the analysis, and to maintain this information throughout compilation until it is needed by some code transformation pass. Dependence information is one of them. When reordering machine instructions that access the memory, for example, it is useful to know about the *certainty* of a memory dependence. If the dependence is certain, the compiler can remove load-store pairs, while if two memory accesses can be proved to be independent, the instructions can be rearranged. Neglecting high level semantic of abstract data structures like lists, binary trees and sparse matrices limits the effectiveness of memory dependence analysis reduces the exploitable parallelism. Hummel et al. [6] classify the structural properties of dynamically allocated, pointer based data structures. These properties may allow the compiler to disambiguate references to pointer-based data structures. This approach requires analyses that need type information about the data structures at level of machine code. In [3] a suite of heuristics for improved static branch prediction is presented that makes use of source-level and variable type information.

Link-time conversion from floating point to fixed point. Due to cost/performance constraints, ASIPs often do not have floating-point hardware. Software emulation is also impractical, because it is too slow. A common compromise between speed and precision is the use of fixed-point arithmetic. One of the features that we intend to implement in our code generator is data type conversion from floating- to fixed-point. Currently, our converter is a source-to-source transformation tool working on a SUIF representation and processing one file at a time [1]. Our converter statically determines the fixed-point format of variables by propagating the type information already available along the data

[3] The generic machine uses an infinite register set model, therefore assigning new registers to a piece of inlined code is straightforward.

flow graph and by applying a set of fixed-point arithmetic rules. When a static function call is encountered, the fixed-point format information can flow in two directions: from a converted actual argument to a formal argument or vice versa.

The current converter can not propagate fixed-point information across separate compilation units through external function calls. To overcome this limitation, we propose to complete floating- to fixed-point conversion of external functions at link-time. This strategy allows to maintain one floating-point set of libraries instead of several fixed-point versions. Moreover, the conversion of a library function can be tailored to the fixed-point format of the call sites of the program being linked. This allows to take advantage of a priori information, like the fact that the result type of several mathematical functions has a limited range and therefore a certain, optimal fixed-point format.

6 Related Work

Wall and Srivastava [12, 11] dedicated much work on optimizations performed at link-time. They found that even simple transformations that do not move or insert code, can greatly improve the code quality, if performed on whole programs. In [12] Wall proposes to delay register allocation until link-time, when all the call sites to a procedure are known and an exact call graph can be computed. This allows to consistently allocate global variables in registers and to remove save and restore code around function calls. This work remarks the difficulty of performing extensive code transformations on a binary format at link time. The author's solution is to encode additional ad-hoc information into special relocation tables of the binary files, to direct the subsequent link-time code transformations. This approach is difficult to generalize.

In a later work [11] the authors propose *OM*, a complete modification system that supports intermodule code optimizations. *OM* translates binary code into a Register Transfer Language and a symbol table. Code transformations are performed on this symbolic representation, which is then translated back to binary. A major shortcoming of this approach is that the source-level information available for optimizations is limited: a substantial part, like type information, has been lost when the binary code was generated. Even the part that can be retrieved requires considerable effort. In practice, *OM* must undo the work done by the compiler system. These work suggest that a structured format that preserves the source-level information could be used to more effectively perform whole-program optimizations.

The idea of exploiting MachSUIF to skip the assembly step has been demonstrated by Greenbaum. In [4] he describes a code generator for experimental Digital Signal Processors that directly translates the MachSUIF representation to binary object files.

7 Conclusions and Future Work

A new strategy to code generation and linkage, superseding the traditional path based on binary tools, has been described. We have motivated its application to ASIP code generation, which would benefit from the source-level information that we maintain throughout compilation. We implemented this approach in our code generator and tested it on a SPARC 20 running Solaris 2.4. The results of linkage demonstrate that our approach is feasible; e.g. the size of MachSUIF binaries of *libgcc* (500KB) and *libc* (4.7MB) libraries are 2.4 and 3.3 times larger than the equivalent binary code on the host machine. This is in part due to the wealth of information not present in a normal binary format, like source-level type information. We consider the increase in code size an acceptable trade-off for the increased flexibility offered by the MachSUIF representation. The approach is applicable to code generators based on MachSUIF for any target architectures.

Future research will focus on applications that take advantage of the new code generation path. Code transformations that require analysis techniques crossing the boundary of single compilation units will be experimented. An interesting application is semi-automatic program conversion from floating- to fixed-point format. We intend to integrate conversion and linkage of floating-point library functions to fixed-point programs, a sort of adaptive library linkage.

References

[1] Andrea G. M. Cilio, Ireneusz Karkowski, and Henk Corporaal. Fixed-point Arithmetic for ASIP Code Generation. In *Proceedings of the 4th conference of the Advanced School for Computing and Imaging*, June 1998.

[2] Henk Corporaal. *Microprocessor Architectures; from VLIW to TTA*. John Wiley, 1997. ISBN 0-471-97157-X.

[3] Brian Deitrich, Ben-Chung Cheng, and Wen-mei W. Hwu. Improving static branch prediction in a compiler. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, October 1998.

[4] Jack Greenbaum. Generating Object Files Directly from SUIF/MACHSUIF Using GNU libbfd.a. In *Proceedings of the Second SUIF workshop*, August 1997.

[5] Stanford Compiler Group. *The SUIF Library*. Stanford University, 1994.

[6] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1994.

[7] Wen-mei W. Hwu and et al. Compiler technology for future microprocessors. *IEEE Transactions on Computers*, 41(12):1625–1640, December 1995.

[8] Johan Janssen and Henk Corporaal. Registers on demand: an integrated region scheduler and register allocator. In *Conference on Compiler Construction*, April 1998.

[9] Clifford Liem. *Retargetable Compilers for Embedded Core Processors*. Kluwer Academic Publishers, 1997. ISBN 0-7923-9959-5.

[10] Michael D. Smith. Extending SUIF for Machine-dependent Optimizations. In *Proceedings of the First SUIF Workshop*, January 1996.

[11] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. Technical Report 6, Western Research Laboratory, Digital Equipment Corporation, December 1992.

[12] David W. Wall. Global register allocation at link time. Technical Report 6, Western Research Laboratory, Digital Equipment Corporation, October 1986.