Compilation of Stack-Based Languages (Abschlußbericht)

M. Anton Ertl Christian Pirker
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
{anton,pirky}@mips.complang.tuwien.ac.at
Tel.: (+43-1) 58801 4474
Fax.: (+43-1) 505 78 38

Abstract

RAFTS is a framework for applying state of the art compiler technology to the compilation of stack-based languages like Forth and Postscript. The special needs of stack-based languages are an efficient stack representation, fast procedure calls, and fast compilation. RAFTS addresses the stack representation problem by allocating stack items to registers such that most stack accesses in the source program are register accesses in the machine language program, and by eliminating most stack pointer updates. To achieve fast calls, RAFTS performs these optimizations interprocedurally and also performs procedure inlining and tail call optimization. Fast compilation is achieved by selecting fast algorithms and implementing them efficiently. Until now we have implemented the basic block part of RAFTS and a part of the work necessary for inlining and interprocedural optimizations.

1 Introduction

Stack-based languages like Forth and Postscript are widely used for embedded control applications (from laser printers to satellites) and for applications involving machine-generated code (e.g., compilers, debuggers, typesetting). The most prominent applications are in the publishing industry, which makes extensive use of Postscript, and in the firmware of SPARC and PowerPC machines, which is encoded in Forth-based FCode to achieve processor independence.

Stack-based languages are distinguished by their heavy use of programmer-visible stacks. Traditionally they are implemented using interpretive techniques. This approach simplifies the implementation, but incurs severe performance penalties.

Most applications would profit from faster implementations of stack-based languages: In embedded control applications and laser printers a more efficient language implementation could provide the same level of performance with cheaper processors (or better performance and functionality with the same processor). In a Display Postscript environment, faster execution means faster graphics. In publishing, sophisticated Postscript programs can block typesetting machines for hours; a faster implementation would improve the utilization of these expensive machines and make their performance more predictable. A fast implementation could also make device drivers written in FCode possible, which would eliminate the costly need to compile and distribute expansion card device drivers for various processors.

A popular avenue to efficiency is the creation of special hardware, which has led to many research-prototype and several commercially available processors that are designed to run Forth efficiently [Koo89]. The code for these stack machines is generated by simple, peephole-optimizing compilers.

Recent research [CU89] shows that unconventional languages can be implemented efficiently on mainstream hardware using aggressive compiler techniques and that the gains of specialized architectures are usually offset by the better manufacturing technology available to widely-used RISCs.

The special needs of stack-based languages are fast stack manipulation and fast procedure calls. Optimizing stack manipulation is especially important for RISC processors, where, for the usual memory-based stack representation, stack accesses cannot be hidden in autoincrement/decrement addressing modes, and each stack access requires two instructions. In addition, special attention has to be paid to preserve the interactiveness of the languages; i.e., compilation has to be very fast to keep compilation time negligible.

This project investigates compiler techniques that address the special needs of stack-based languages. Forth is used as the subject of experimentation due to its simplicity. Postscript can be compiled by combining the techniques explored in this project with techniques for optimizing dynamically type-checked languages [CU89] and techniques for reducing the amount of dynamic binding.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: Final report for FWF project P11231

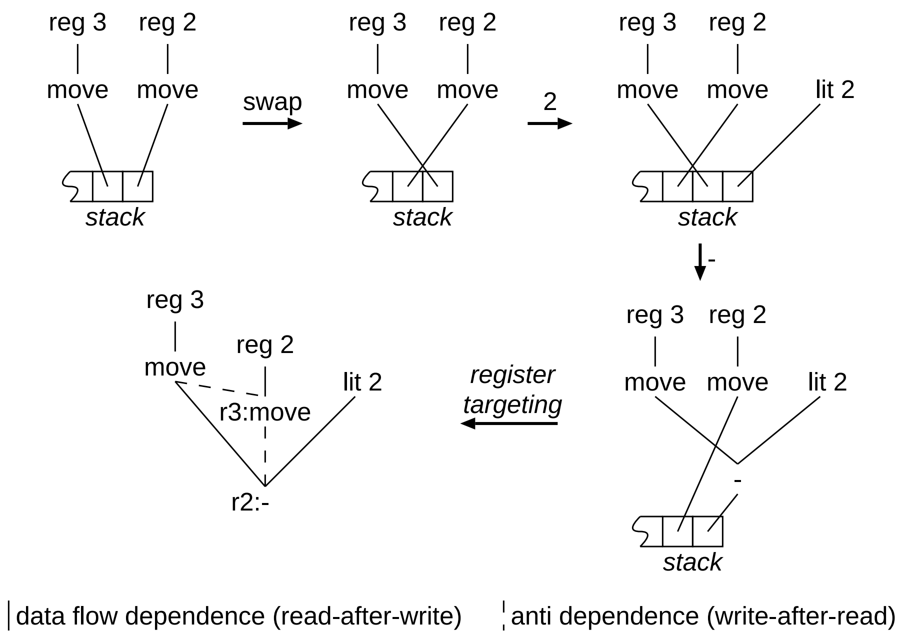

Figure 1: Building the intermediate instruction graph for the sequence `swap 2 -` by symbolic execution. At the basic block boundaries the top of stack resides in register 2 and the next stack item in register 3.

While our research focuses on stack-based languages used by humans, it addresses problems that are also relevant for the JavaVM and similar intermediate representations: the JavaVM needs an efficient stack representation, fast calls (typical JavaVM code frequently executes calls), and very fast compilation (because the compile time is part of the program execution time experienced by the user).

RAFTS is a framework for applying state of the art compiler technology to the compilation of stack-based languages. Section 2 gives a short overview; Section 3 describes our results for the project up to now. If you would like a more detailed account of the results, read our papers [EP96, EP97].[1]

2 RAFTS Overview

The basic principle in our project is to map stack items to registers, in contrast to the conventional memory-based stack implementation.

In some cases it is not possible to map all stack items to registers, because stacks are unlimited in size, and because the stack depth sometimes cannot be determined at compile time. In these cases we try to minimize the resulting memory accesses and stack-pointer updates.

2.1 Basic Blocks

The compiler deals with basic blocks by transforming the stack-based code into data dependence graphs (i.e., data flow graphs with additional edges that represent anti dependences and memory dependences): It symbolically executes the stack-based code; for the symbolic execution, the operations work on a stack containing dependence graph nodes, and most operations build additional nodes (see Fig. 1). The nodes in the resulting graph are intermediate instructions, so we call it the intermediate instruction graph.

Then the compiler applies standard basic block code generation techniques (see Fig. 2): it uses tree parsing for instruction selection (resulting in a data dependence graph with machine instructions as nodes, the machine instruction graph), list scheduling for instruction scheduling, and a simple local register allocation technique.

The basic block part deals with the values on the stack within basic blocks very effectively, and allocates all of them to registers (unless it runs out of registers, which never happens in practice).

2.2 Beyond Basic Blocks

Values on the stack on basic block boundaries pose more problems. Since the main reason for basic block boundaries in stack-based code written by humans are calls and returns[2], the effectiveness of intraprocedural register allocation is limited; the

[1] You can find them through http://www.complang.tuwien.ac.at/projects/rafts.html.

[2] 26%-34% of the virtual machine instructions executed in a Forth interpreter for typical programs are calls or returns.

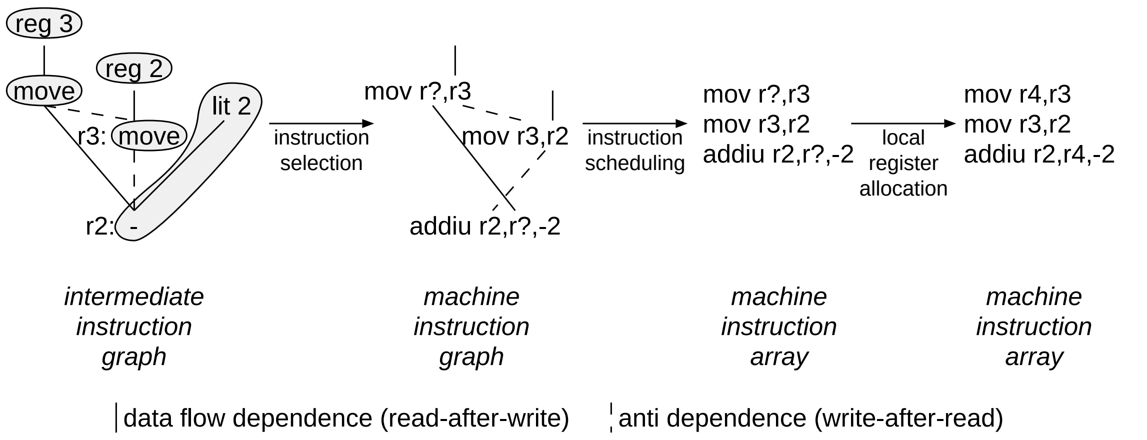

Figure 2: Code generation (for the MIPS architecture).

processor would spend a lot of time saving and restoring registers at procedure call boundaries. Interprocedural register allocation and/or aggressive inlining is necessary to make effective use of the register set.

Another problem is the stack pointer updates. The basic block code generation reduces the number of stack pointer updates to at most one per stack and basic block. It is possible to reduce the number much more. E.g., in procedures where all stack items are allocated to registers, no stack pointer update is needed at all. Like register allocation, stack pointer update minimization has to be performed interprocedurally to achieve a significant effect.

We have not yet completed the work in this area.

3 Results up to now

Currently we have a working compiler for the MIPS architecture that can compile itself repeatedly (until it runs out of memory); it runs on SGI machines under Irix and on DecStations under Ultrix. This compiler implements the basic block part of RAFTS; we have also incorporated some work that will be useful for interprocedural register allocation, stack pointer update minimization and inlining (e.g., the linear intermediate code and on-demand code generation), but the actual interprocedural optimizations are not yet implemented. We are about as far relative to the amount of work invested as we estimated in our original project proposal; because only a part of the originally proposed staff was granted, only a part of the work is complete.

For a detailed description of our results read [EP96, EP97][3]. Here we highlight some of the findings which we did not expect at the start of the project:

3.1 Code Generation

To our surprise, we even discovered something new in general, language-independent code generation:

- For basic block instruction scheduling we use the most popular approach: list scheduling. It requires finding all instructions without predecessors in the graph, at every step. We originally thought that we would have to use a complex additional data structure to make it efficient, until we realized that just adding reference counts to the existing dependence graph would suffice, if we use backwards list scheduling instead of forward list scheduling.

- In local register allocation we have to deal with the problem of register shuffling: Implementing simultaneous multiple assignment with a sequence of simple assignments. The optimal solution is quite complex (and slow), the simple solution can generate many superfluous assignments in the worst case. We found a method for dealing with register shuffling that is only slightly (3 lines of code) more complicated than the simple solution, yet produces only half as many superfluous moves in the worst case; it is hardly slower when it does not save moves, and faster when it does.

- A naïve implementation of the phases dependence graph construction, instruction selection, instruction scheduling, local register allocation, and code emission requires nine passes. They can be integrated into four passes (see Fig. 3), resulting in faster code generation. The use of backwards instruction scheduling plays an important role here; forward instruction scheduling would require two additional passes

[3] Our papers are available at http://www.complang.tuwien.ac.at/papers/.

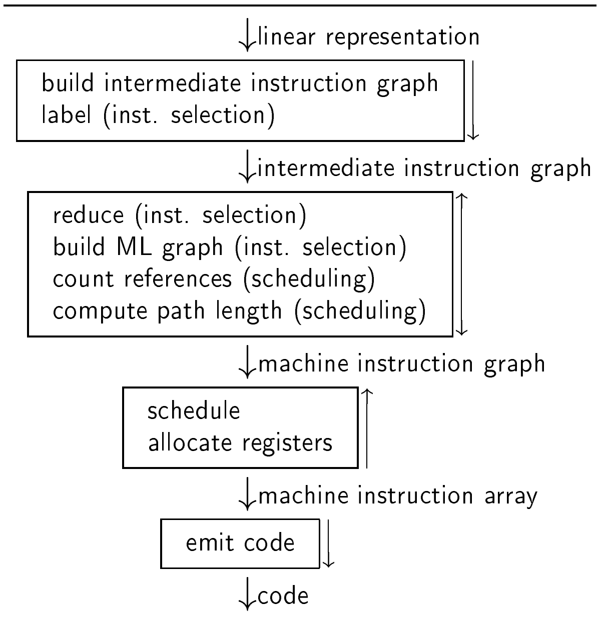

Figure 3: Phases, passes and data structures in our code generator. ↓ indicates a forward pass (i.e., with the data flow), ↑ indicates a backward pass (against the data flow), ↕ indicates a recursive graph walk starting at the bottom.

(in addition to more complex data structures, see above).

- We extend the tree parsing approach to code selection to DAGs. The problem with this extension is that, unlike tree parsing, it does not necessarily guarantee optimality. We have done some work on showing for which code selection grammars it is optimal (to be published); it is optimal for the code selection grammar we use.

We are quite pleased with the resulting code generator and would also use this approach for other compilers, unless the requirements differ greatly.

3.2 Intermediate Representation

For interprocedural optimizations, in particular for inlining, we have to store the code in some intermediate form before generating code. Originally we wanted to use an extended form of the intermediate instruction graphs; however, this representation would have made code replication (for inlining, and possibly for other optimizations) quite cumbersome. Therefore we use a linear intermediate representation.

The original feature of this intermediate representation is that it is not just a data structure, but threaded code, which can be executed quickly; upon execution it generates the intermediate instruction graphs for the piece of source code it represents, and at basic block ends it calls the code generator to produce machine code. Threaded code is also easy (and quick) to generate. We use a stack-based linear intermediate representation; however, threaded code could also be used for other kinds of intermediate languages (e.g., quadruples).

3.3 Forth

We also found out some properties of Forth and its implementations, and we discuss two of them here:

- In traditional, threaded code implementations of Forth, basically all words[4] are compiled with the same simple routine; in native code compilers each word requires a special routine.

  Several implementors tried to solve this problem with a technique called `state`-smartness; unfortunately, this solution does not work correctly in some cases.

  We evaluated three correct solutions, and decided to use the *intelligent* ***`compile,`*** approach, which is relatively easy to implement and, in some cases, produces better code than the other solutions.

- We bootstrap our compiler on a threaded code system. To minimize the cross-compilation type of complications, we did a full integration of native code and threaded code. It is possible to call native code from threaded code and vice versa. We had to solve some interesting problems to provide this flexibility.

3.4 Empirical Results

All timings reported here were taken on a DecStation 5000/200 (25MHz R3000) with 40MB RAM.

3.4.1 Small Benchmarks

For a few small benchmarks we can compare our compiler with both, a C compiler and Gforth, a threaded-code system. Figure 4 shows the results. The code generated by RAFTS is between slightly faster and twice as slow as the code produced by GCC. The main reason for the slowdown is accesses to stack items in memory, and should be eliminated by global register allocation. The speedup over Gforth, a threaded code system, is 3.32–4.46 for these benchmarks.

3.4.2 A Larger Benchmark

In this section we use self-compilation of RAFTS as a large *run-time* benchmark. We compare RAFTS with Gforth, a threaded code system; i.e.,

[4]Forth *words* correspond to Postscript operators and to functions and procedures in other languages.

	GCC -O2	RAFTS	Gforth	GCC -O2 abs. time
sieve	1.00	1.69	5.85	8.7s
bubble	1.00	1.97	6.53	8.7s
matmul	1.00	1.28	5.73	7.4s
fib	1.00	0.89	3.69	14.1s

Figure 4: GCC -O2 vs. RAFTS vs. Gforth for small benchmarks (relative time)

relative time		producing threaded code	producing native code
running	threaded code	1.00	4.13
running	native code	—	2.91

Figure 5: Relative compile times of different compilers executed with different execution techniques (compiling the RAFTS prototype; no stack items in registers)

we compare the run-time of RAFTS in (RAFTS-compiled) native code with the run-time of RAFTS in threaded code; the input data for this benchmark was RAFTS.

The speedup over Gforth is 1.41 (for one stack item in a register), less than for the smaller benchmarks, and somewhat disappointing. One of the reasons is that even the RAFTS-compiled version of RAFTS executes a significant number of threaded-code words, mainly for I/O and dictionary lookup; and calling threaded code words from native code involves more overhead than in threaded code. We estimate that about a third of the run-time of RAFTS-compiled RAFTS is spent in threaded code words. We think that another reason for the low speedup is that most basic blocks in RAFTS are very short (0-2 non-stack-accessing instructions), reducing the effectiveness of our basic block optimizations and increasing the proportion of stack access instructions; adding interprocedural register allocation should be especially effective for this code.

3.4.3 Compilation Speed

We also use the compilation of RAFTS for comparing the compilation speed of RAFTS and Gforth; here we compare the compile-time of a RAFTS-compiled RAFTS with the compile-time of Gforth's compiler, which produces threaded code. The input data for the compilers was, of course, RAFTS.

Figure 5 shows the results (it also shows the results for the compile-time with RAFTS running in threaded code). RAFTS compiles about three times slower than Gforth (if it generates code with no stack item in registers); a slowdown is to be expected, because RAFTS has to perform the same I/O and dictionary lookup tasks as Gforth (and currently most of these tasks actually use the same threaded code), but has a much more complicated and time-consuming code generator[5].

	items	cycles/item
compile-time (s)	9.89	25000000
lines	6411	38566
linear IR words	15940	15511
IL nodes	32520	7603
ML nodes	24687	10015
native code cells	26984	9162

Figure 6: Size of the compiler and compilation speed (absolute and in cycles/item); this is not a split of the time into various subtasks.

Figure 6 shows some numbers on the size of RAFTS and on the absolute speed. We think that with more tuning there is still an order-of-magnitude improvement in compile-time possible.

References

[CU89] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, 1989.

[EP96] M. Anton Ertl and Christian Pirker. RAFTS for basic blocks: A progress report on Forth native code compilation. In *EuroForth '96 Conference Proceedings*, St. Petersburg, Russia, 1996.

[EP97] M. Anton Ertl and Christian Pirker. The structure of a Forth native code compiler. In *EuroForth '97 Conference Proceedings*, pages 107–116, Oxford, 1997.

[Koo89] Philip J. Koopman, Jr. *Stack Computers.* Ellis Horwood Limited, 1989.

[5] The "code generator" of a threaded code system just stores the value found in the dictionary into memory.