DISSERTATION

Implementation of Stack-Based Languages on Register Machines

ausgeführt zum Zwecke der Erlangung des akademischen
Grades eines Doktors der Technischen Wissenschaften

eingereicht an der Technischen Universität Wien
Technisch-Naturwissenschaftliche Fakultät

von

Martin Anton Ertl, Wiedner Hauptstraße 141/1/7, 1050 Wien
Matr. Nr. 8525644
geb. am 21. August 1967 in Wien

Wien, im April 1996

Abstract

Languages with programmer-visible stacks (stack-based languages) are used widely, as intermediate languages (e.g., JavaVM, FCode), and as languages for human programmers (e.g., Forth, PostScript). However, the prevalent computer architecture is the register machine. This poses the problem of efficiently implementing stack-based languages on register machines. A straight-forward implementation of the stack consists of a memory area that contains the stack items, and a pointer to the top-of-stack item.

The basic optimizations explored in this thesis are: Caching the frequently-accessed top-of-stack items in registers reduces stack access overhead, and combining stack-pointer updates eliminates most of them.

This thesis examines these optimizations in the context of three basic implementation techniques:

- For (virtual machine) **interpreters**, I regard the stack cache in the registers as finite state machine, where the execution of a virtual machine instruction performs a state transition; there are specialized implementations of the virtual machine instructions for each state.

- My **native-code compilation** technique transforms the programs into standard compiler data structures; then state-of-the-art compiler technology can be applied for optimization and code generation. In particular, stack items are represented by pseudo-registers, which register allocation will (usually) put into machine registers; stack pointer updates are executed symbolically, i.e., at compile time.

- For **translation to C**, I emphasize simplicity; the optimizer of the C compiler takes care of the complex problems. The translator just has to represent stack items as local C variables, and the C compiler will keep them in registers. As with native-code compilers, (usually) no stack pointer updates at run-time are necessary.

For interpreters, the optimizations eliminate about two real machine instructions per virtual machine instruction, resulting in speedups of 17%–31% (for Forth on a DecStation 3100). The techniques presented here for native-code compilation and translation to C achieve a speedup factor of 1.3–3 over traditional native code compilers and more than 3 over a straight-forward translator to C (for Forth on a 486DX2/66).

Contents

Chapter 1

Introduction

Languages with programmer-visible stacks (stack-based languages) are used widely, often as intermediate languages (e.g., JavaVM, Smalltalk byte-code, PostScript, FCode), but also as languages for human programmers (e.g., Forth, PostScript, Pop, the BibTeX style language).

The prevalent processor architecture in high-performance computers (from supercomputers down to PCs) is the register machine, i.e., an architecture that provides several (8–32) general-purpose registers.

Clearly, the efficient implementation of stack-based languages on register machines is an important task. This thesis explores this topic for several implementation techniques: interpreters (Section 2), compilers producing native machine code (Section 3), and translation to C code (Section 4).

A straight-forward implementation of the stack consists of a memory area that contains the stack items, and a pointer to the top-of-stack item (see Fig. 1.1). Each operation loads the operands from memory with an indexed access through the stack pointer, stores the results to the stack in the same way, and changes the stack pointer, so that it points to the new top-of-stack item.

My main contribution is the application of the following ideas in the context of the different implementation techniques: frequently-accessed top-of-stack items are kept in registers, eliminating memory accesses; and stack pointer updates are combined, eliminating most of them. A more detailed

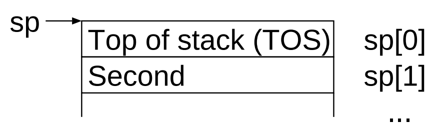

Figure 1.1: A straight-forward implementation of a stack and how to access it in C

account of my contributions can be found in each chapter.

The rest of this chapter examines the possibility of employing a stack architecture (Section 1.1), compares the implementation techniques discussed in this thesis (Section 1.2), takes a closer look at stack-based languages (Section 1.3) and shows that (most) register architectures offer enough registers to make the techniques explored in this thesis worthwhile (Section 1.4); finally, it gives a short overview of the assembly languages used in this thesis (Section 1.5).

1.1 Processor architecture

Stack architectures have been proposed for executing stack-based (and other) languages efficiently [Bla77, HFWZ87, HL89, Koo89]. Unfortunately, it is often not possible to choose the architecture. Even if you are allowed to choose the processor, there may be no stack architecture in the market segment you are interested in. In particular, the high-performance market (PCs and up) is monopolized by register architectures.

As a last resort, you might try to build a stack architecture processor yourself; however, research in architectures and compilers for other unconventional languages [Ung87, CU89, KB92] has shown that such 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 register architectures. This thesis shows that this is also valid for stack-based languages.

Stack architectures (and other non-register-architectures, e.g., accumulator architectures) still have a niche in special applications and in the embedded control market; however, in the long run the improved fabrication technology will lead to a situation where the advantages of such architectures (smaller processor core, smaller code) are no longer significant. The market share of register architectures (and the application domain of the techniques presented here) will increase.

1.2 Implementation techniques

Why do I explore three different implementation techniques? Is there not one true way of implementing programming languages? Unfortunately, the answer is no; each technique has its advantages and disadvantages (some are listed in Fig. 1.2). For example, an interactive language like Forth, that can

	interpreter	native code compiler	translator to C
execution speed	low	high	high
compilation speed	high	high?	low
retargetability	cheap	expensive	cheap
interactivity	yes	yes	no

Figure 1.2: Advantages and disadvantages of various language implementation techniques

compile new code at run-time, cannot be translated completely into C, which enforces a strict separation between compilation and run-time.[1]

1.3 Stack-Based Languages

1.3.1 Overview

Languages with programmer-visible stacks are used as general and special-purpose programming languages, they are popular as intermediate languages for compilers and they are very popular as machine-independent executable program representations.

A language has a *programmer-visible stack*, if it is hard or impossible to describe its semantics without describing a stack. With regard to implementations this means that a straight-forward implementation would incorporate a stack implementation like the one shown in Fig. 1.1. Here are some examples:

Programming languages

Forth is used in many fields; it is especially popular in embedded control applications.

PostScript is used primarily in typesetting and other display purposes; its versatility is shown by an application in a debugger [RH92]. Because the majority of PostScript code is written by programs, it can also be regarded as intermediate language.

Pop is primarily used by the British AI community; it offers infix notation for the benefit of those who are uncomfortable with postfix.

[1]Dynamic linking allows hacking around this separation, but the result is not very portable (dynamic linking is not universally supported) and does not feel interactive.

RPL (reverse polish LISP) is the language of the HP-48 calculator. It is a run-time type-checked language with many mathematical data types and Forth-like syntax.

The BibTeX **style language** is a special-purpose language for processing BibTeX databases.

Intermediate languages for compilers

UCSD p-code is used in the UCSD system, an operating system best known for its Pascal compiler. P-code was either interpreted or compiled to native code.

EM is the intermediate language of the Amsterdam Compiler Kit (ACK) [TvSKS83]. It is usually translated to native code.

Elisp byte-code is a code for Emacs Lisp programs, that can be interpreted more quickly than the list representation, and can be exchanged between machines without recompilation.

Smalltalk-80 byte-code [GR83, Kra83] is the intermediate language of the Smalltalk-80 system.

Machine-independent executable program representations

JavaVM [Sun95] is a byte-code for transmitting WWW applets. Designed for the Java language, but not limited to it.

Fcode is a machine-independent binary representation of Forth. The firmware in SBus and PCI cards (Open Firmware) is represented as Fcode for CPU independence.

These categories overlap somewhat. E.g., PostScript and Elisp byte-code could be just as well regarded as belonging to the category "machine-independent executable program representations".

Stack-based languages are used as programming languages, because

- they are conceptually simple. There is no need to deal with many different concepts such as statements, r-expressions, l-expressions, sequence points[2], with syntax definitions, that even in BNF notation need several pages, etc.

[2] These terms come from C terminology; other languages in the Algol family use different terms.

- they make metaprogramming (programs that deal with programs) easy. This is another consequence of their conceptual simplicity. In many stack-based languages it is possible to build programs at run-time and execute them immediately.
- they offer a higher quality of extensibility. There is no syntactical (and, in many cases, no internal) difference between a built-in and a user-defined operation. In contrast, in the languages in the Algol family built-in operators differ syntactically from user-defined functions.

However, their popularity as programming languages is limited, because

- the syntax is unusual. Everybody learns infix syntax for arithmetic in school. Learning postfix requires extra effort.
- parameters are passed implicitly. This can make programs harder to read, if they are not written in good programming style. In particular, programmers coming from an Algol language tend to write longer definitions (procedures) than is good style in stack-based languages.
- "everybody programs in C (C++)".

Some of the reasons for their popularity as intermediate languages are:

- It is easy to generate code for them from most languages.
- They can be implemented efficiently as interpreters (see Section 2.2.1).
- They impose fewer arbitrary limits (e.g., a limited number of registers)
- Programs in these languages can be represented compactly as byte code (for permanent storage or transfer over a network).

One important property of a language with regard to its implementation is, whether is is type-checked at run time or not[3]. About half of the languages mentioned above (PostScript, Pop, RPL, the BibTeX style language, Elisp byte-code, and Smalltalk-80 byte-code) are dynamically type-checked, the others are not. This work does not deal with run-time type-checking, because run-time typechecking is neither universal in nor specific to stack-based languages. The techniques developed for implementing it in other languages (e.g., SELF [CU89]) should work for stack-based languages, too.

[3]For the present work, it does not matter whether the language is type-checked at compile-time or not at all; compile-time type-checking belongs to the duty of compiler front-ends, which are not in the scope of the present work.

1.3.2 Forth

I will use Forth [ANS94, RCM93] as a representative for stack-based languages in this thesis. Forth offers several advantages:

- It is simple. Those of its properties that are specific to Forth, do not pose hard implementation problems.
- It offers direct access to the stack-based language (in contrast to systems that use the stack-based language only as intermediate code).
- It is interactive, further facilitating experimentation.

Other properties important for the present work are:

- Forth is usually[4] *not type-checked.*[5] In this respect Forth is similar to many intermediate languages and assembler.
- It operates on *machine words* (*cell*s in Forth terminology). The items on the stacks are cell-sized; only things that fit into a cell are treated first-class. Again, Forth is similar in this respect to many intermediate languages, assemblers, and languages such as Pascal.

Syntactically, a Forth program consists of a sequence of *word*s[6], i.e., sequences of non-whitespace characters separated by white space. A Forth *colon definition* corresponds to a C function. E.g.,

```
: squared
  dup * ;
```

defines the word `squared` with the stack effect `n1 -- n2`. When `squared` is executed, it executes first `dup`, i.e., it pushes another copy of the top-of-stack on the stack; then it executes `*`, i.e., it multiplies the two copies of the input parameter and produces the square; then it returns.

[4][ANS94] specifies type rules, and there has been some work on compile-time type checking in Forth [SK93].

[5]Some people call languages without type-checking (and sometimes even languages with run-time type-checking) *untyped.* This term is inappropriate: Forth works on data, and each datum has a type (an interpretation). Applying the wrong operation to a datum is a type error, even if neither the compiler nor the run-time system will catch it; the programmer is responsible for writing type correct programs.

[6]The use of *word* to denote a program element makes it necessary to use a different term for the basic data element, namely *cell.*

1.3.3 Stack effect notation

Stack-based languages use a (semi-)formal notation for specifying the stack contents and the stack effect of operations.

The *stack contents* are specified as a sequence of stack items, with the top-of stack rightmost. E.g., `a b c` specifies that `c` is the top-of-stack item, `b` is the second stack item, and `a` is the third item. The rest of the stack (and the stack depth) is not specified. The order of the stack items in the specification corresponds to the order, in which you type them in if you want to produce these stack contents.

The *stack effect* of a word is specified by specifying the stack contents before the word is executed and the stack contents afterwards, separated by `--` (in Forth; the PostScript convention differs in this respect by using the operator name as separator). Those parts of the stack that are not specified stay the same during the execution of the word. E.g., the stack effect for `swap` is `a b -- b a`.

The stack effect of two words can be combined in the intuitive way to compute the combined stack effect. Formally:

$$\begin{aligned} x_1...x_k \text{ -- } y_i...y_l \times y_1...y_l \text{ -- } z_1...z_m &= y_1...y_{i-1}x_1...x_k \text{ -- } z_1...z_m \\ x_1...x_k \text{ -- } y_1...y_l \times y_i...y_l \text{ -- } z_1...z_m &= x_1...x_k \text{ -- } y_1...y_{i-1}z_1...z_m \end{aligned}$$

E.g., in the `squared` example, `dup` has the stack effect `x -- x x`, `*` has the stack effect `n1 n2 -- n3`, so the combined stack effect (i.e., the stack effect of `squared`) is `n1 -- n2`.

Stack effects, augmented with typing rules, form an algebra that can be seen as specifying a syntax [Pöi94]. This formalism is better suited for describing the syntax of stack-based languages than context-free grammars.

1.3.4 Specific Forth properties

Forth has some properties that are relevant for the present work, where Forth differs from some of the other stack-based languages:

- Forth has a separate *return stack* for holding return addresses; return addresses do not get in the way when the parameters are accessed. In Forth the return stack is also programmer-visible, and is sometimes used by programmers for holding temporary data. The stack normally used for data is called *data stack*.

- ANS Forth specifies (state-less) control flow with seven words that communicate (at compile-time) through a *control-flow stack* (see Fig. 1.3). All (state-less) control structures can be specified with these words

Word	Stack effect	Meaning
`IF`	-- orig	conditional forward branch
`AHEAD`	-- orig	unconditional forward branch
`THEN`[7]	orig --	target of forward branch
`BEGIN`	-- dest	target of backward branch
`UNTIL`	dest --	conditional backward branch
`AGAIN`	dest --	unconditional backward branch
`CS-ROLL`	... n -- ...	control-flow stack manipulation

Figure 1.3: Forth's basic control structure words. A dest item on the control-flow stack represents the target of a backward branch, an orig item represents the source of a forward branch.

[Bad90]. In other words, these words have the same expressive power as the combination of unconditional and conditional branches present in many intermediate and assembly languages. The advantage of this form of expressing control-flow is that it is easier to implement (no label table necessary), that structured (i.e., goto-less) control flow is easier to translate to these primitives, that there are no name clash problems (e.g., on macro expansion), and that it facilitates control flow analysis.

I will discuss the consequences of these properties in this thesis, but I will also point out what is Forth-specific, and how the results for other languages would differ from the results for Forth.

1.3.5 Program characteristics

The following characteristics of typical programs are important for the implementation:

- Forth code written by human programmers has an extremely *high call frequency*. In these programs every third or fourth dynamically executed word performs a call or a return (see Fig. 2.22). This is probably

[7] In structured code `THEN` terminates an `IF`-structure. According to *Merriam-Webster's School Dictionary*, *then (adv.)* has the following meanings:

> ... 2b: following next after in order ... 3d: as a necessary consequence (if you were there, then you saw them).

Forth's `THEN` has the meaning 2b, whereas `THEN` in Algol and its offspring has the meaning 3d. Many Forth programmers use `ENDIF`[8] instead of `THEN` to terminate `IF`-structures.

[8] `ENDIF` is easy to define: `: ENDIF ( orig -- ) POSTPONE THEN ; immediate`

also the case for programs written by humans for other stack-based languages, but compiler-generated code probably exhibits a lower call frequency.

- The stack depth for each point in the definition (relative to the stack depth at the beginning of the definition) can usually be determined at compile-time. An unknown stack depth can arise at control-flow joins, if the stack depths of joining control-flow edges differ. Intentional unknown stack depths are so rare that [Tev89] has proposed that the compiler should produce a warning on detecting an unknown stack depth, because it probably is due to a programming error. This is probably also true for other stack-based languages, both for code written by humans, and especially for compiler-generated code.

1.4 Stack usage characteristics

We want to improve the performance of implementations of stack-based languages by keeping stack items in registers and by reducing stack pointer updates. Will this actually have a significant effect? How many registers do we need, and how much improvement will we see?

1.4.1 Theoretical work

A mathematical model for the stack behaviour would be very useful: It could be used to predict the performance of various implementations without requiring tedious (and, possibly, non-representative) empirical measurements; and, more importantly, it would allow the application of analytical methods to derive an optimal implementation instead of using trial and error.

In [HS85], Hasegawa and Shigei propose the *random walk model* of stack behaviour and they use this model to design a hardware stack caching algorithm. The random walk model assumes that, at every point in program execution, the probability of performing a push and a pop are the same, irrespective of previous events. In other words, if you have a coin that says *push* on one side and *pop* on the other, throwing the coin repeatedly would produce a random walk model sequence. Hasegawa and Shigei did not evaluate the random walk model empirically.

[Hay89, Ert95] present some empirical data and draw the conclusion that the random walk model does not match the observed behaviour. Let us look at some of the predictions of the random walk model. It assumes a stack cache that can cache the first s stack items. It predicts

- that, if the cache is half-full, the probability of cache overflow and cache underflow (as next event requiring external access) is equal, irrespective of previous events. I observed the behaviour of four real-world programs (see Fig. 2.22) after an overflow, with the overflow resetting the cache to half-full. I found that only one of the programs (*gray*) performed another overflow in this situation, and then only 10 out of 279 times (the random walk model predicts 139.5).

- that, resetting the cache to half-full after an overflow minimizes the number of overflows, and (if all ways of handling overflows cost the same) the total cost. I observed that it is optimal to keep the stack rather full after an overflow (see Fig. 2.24).

Hayes [Hay89] explains this by taking a close look at loops: In a loop the stack depth returns at the end to the depth it had at the beginning (unless the loop produces an unknown stack depth, which is rare (see Section 1.3.5)). Because most of the time is spent in some loop or other, we should see that the stack depth oscillates around some average.

Moreover, definitions usually have a stack effect that does not change the stack depth much. So, at the end of the definition the stack depth has to come back close to the entry depth. Maybe one day we can find a better model based on these observations.

1.4.2 Empirical work

Hardware stack machines have successfully used small register files for keeping some of the top-of-stack caches. The most radical approach is the Transputer: It provides only a stack of depth three, all of them in registers. The HP3000 performed well with 4 top-of-stack registers [Bla77], the Tandem Nonstop architecture used eight registers [AS92]. The FRISC group eliminated almost all memory accesses with stack caches of 16 registers (with the additional restriction that the top 4 items always must reside in registers) [HFWZ87, Hay89, HL89].

My measurements show that a few registers are sufficient to capture the majority of data stack accesses in Forth. Each additional register approximately halves the number of memory accesses (up to 14 registers, then the decrease slows down). I expect similar benefits for keeping return stack items in registers.

1.5 Assembly languages

Some examples in this thesis are written in assembly languages for the MIPS and Intel 386 architectures. These architectures are representative of the RISC and CISC factions of the register architectures, respectively.

The MIPS architecture is a load/store architecture. Operations (like `addu`) have three operands; all of them must be registers. In MIPS [KH92] assembly language syntax register n is denoted by $\$n$, the destination operand of an instruction is usually the leftmost register, and comments start with `#`.

The Intel 386 (and its offspring) has a 2-address architecture (one of the operands can reside in memory). I use the AT&T syntax used by `gas` and other UNIX assemblers: The destination operand (which usually also serves as a source operand) is on the right, register names start with `%`.

Chapter 2

Interpretation

The major advantages of interpreters over compilation to native code are simplicity and portability. Their major advantages over the generation of C code are compilation speed and flexibility (e.g., to generate additional code at run-time). Interpreters are still the dominant implementation method of general-purpose languages like Prolog, Forth and APL, probably the majority of special-purpose language implementations are interpreters, and they are even used in special implementations of traditionally compiled languages like C.

In recent years many questions about interpreters have been asked in the Usenet newsgroup `comp.compilers`. Efficiency was a major concern; another frequent question is whether to use a stack or a register architecture for the virtual machine.

This chapter first discusses interpreter construction and general efficiency issues (Section 2.1). In Section 2.2, I answer the question, which virtual machine architecture should be used. Then I present several methods for caching stack items in real machine registers: Either there is only one cache state (Section 2.3.1), or the interpreter (Section 2.4) or the compiler (Section 2.5) keeps track of the cache state. Finally, I show some empirical results (Section 2.6).

My main original contributions are the compiler-based static stack caching technique, the discussion of different stack cache organizations, the empirical evaluation, and the analysis of one-state stack caching. The material in this chapter first appeared in [Ert94b, Ert95], and a little bit in [Ert93].

A note on terminology: unless otherwise noted, the terms *instruction* and *primitive* refer to virtual machine instructions, *cache* refers to the stack cache implemented in software, and the *compiler* is the program that generates the virtual machine code.

In this chapter, I often refer to machine language concepts: *registers*,

```
typedef void (* Inst)();

void add(Inst *ip, int *sp /* other regs */)
{
  sp[1] = sp[0]+sp[1];
  (*ip)(ip+1, sp+1 /* other registers */);
}

Inst program[] = { add /* ... */ };
```

Figure 2.1: Direct threading in C using tail calls

(memory-to-register) *load*s, (register-to-memory) *store*s and (register-to-register) *move*s. However, for portability reasons interpreters are usually written in portable languages like C. Good C compilers allocate (scalar) local variables into registers (if there are enough). Moves correspond to assignments between local variables, stores correspond to assignments to anything but local variables (e.g., an assignment to an array element), and loads correspond to (r-value) accesses to anything but local variables. For a more detailed discussion of the correspondence of C and machine code, read Section 4.

2.1 Interpreter efficiency

Since we are interested in efficiency, we limit the discussion to virtual machine interpreters, and will not discuss, e.g., syntax tree interpreters. The interpretation of a virtual machine instruction consists of three parts:

- accessing arguments of the instruction
- performing the function of the instruction
- dispatching (fetching, decoding and starting) the next instruction

The first and third part constitute the interpreter overhead.

2.1.1 Instruction dispatch

The most efficient method for fetching, decoding, and starting the next primitive is direct threading [Bel73]: Instructions are represented by the addresses of the routine that implements them, and instruction dispatch consists of

```
typedef enum {
  add /* ... */
} Inst;

void engine()
{
  static Inst program[] = { add /* ... */ };

  Inst *ip;
  int *sp;

  for (;;)
    switch (*ip++) {
    case add:
      sp[1]=sp[0]+sp[1];
      sp++;
      break;
    }
}
```

Figure 2.2: Instruction dispatch using `switch`

fetching that address and jumping to the routine.[1] Unfortunately, direct threading cannot be implemented in ANSI C and other languages that do not have first-class labels and do not guarantee tail-call optimization (Fig. 2.1 shows how direct threading would be implemented in C using tail-calls).

Two methods are usually used in C: a giant switch (Fig. 2.2) or calls (Fig. 2.3). In the first method instructions are represented by arbitrary integer tokens, and the switch uses the token to select the right routine; in this method the whole interpreter, including the implementations of all instructions, must be in one function. In the second method every instruction is a separate function; this method is actually quite similar to direct threading (it just uses calls instead of jumps), so I call it direct call threading. Figure 2.4, 2.5 and 2.6 show MIPS assembly code for the three techniques (direct call threading needed a little source code twisting to get reasonable scheduling). Fig. 2.7 shows the overhead of these techniques in cycles on two processors,

[1] Forth has been traditionally implemented using indirect threading [Dew75, Kog82], which performs an additional indirection. I.e., an instruction is represented by an address of a cell that contains the address of the routine. This is used for eliminating inline arguments; e.g., instead of having a call instruction with an inline argument, the instruction points to a cell in front of the called word and that cell points to the call routine; direct threaded Forth implementations usually emulate this structure.

```
typedef void (* Inst)();

Inst *ip;
int *sp;

void add()
{
  sp[1]=sp[0]+sp[1];
  sp++;
}

Inst program[] = { add /* ... */ };

void engine()
{
  for (;;)
    (*ip++)();
}
```

Figure 2.3: Direct call threading

```
lw   $2,0($4) #get next instruction, $4=inst.ptr.
addu $4,$4,4  #advance instruction pointer
j    $2       #execute next instruction
#nop          #branch delay slot
```

Figure 2.4: Direct threading in MIPS assembly

the R3000, and the more deeply pipelined R4000. The overhead varies depending on how many delay slots can be filled; usually it will be at the lower bound.

The execution time penalty of the switch method is caused by a range check, by a table lookup, and by the jump to the dispatch routine generated by most compilers. The call method does not look so slow, but it is usually even slower than the switch method: Every virtual machine register, e.g., instruction and stack pointer, has to be kept in global or static variables. Most C compilers keep such variables in memory, causing at least a load and/or store for every virtual machine register accessed in a primitive. In the switch method virtual machine registers can be kept in local variables, which are translated into real machine registers by good compilers.

Fortunately, there is a widely-available language with first-class labels: GNU C (version 2.x); so, direct threading can be implemented portably (see

```
$L2: #for (;;)
 lw    $3,0($6) #$6=instruction pointer
 #nop
 sltu  $2,$8,$3 #check upper bound
 bne   $2,$0,$L2
 addu  $6,$6,4  #branch delay slot
 sll   $2,$3,2  #multiply by 4
 addu  $2,$2,$7 #add switch table base ($L13)
 lw    $2,0($2)
 #nop
 j     $2
 #nop
 ...

$L13: #switch target table
 .word   $L12
 ...

$L12: #add:
 ...
 j   $L2
 #nop
```

Figure 2.5: Switch dispatch in assembly

```
add:
 ...
 j    $31  #return

engine:
 ...
$L3:
 lw   $2,ip #instruction pointer
 #nop
 lw   $4,0($2)
 addu $3,$2,4
 jal  $31,$4 #call $4
 sw   $3,ip  #delay slot
 j    $L3
 #nop
```

Figure 2.6: Direct call threading in assembly

	R3000	R4000
direct	3–4	5–7
switch	12–13	18–19
call	9–10	17–18

Figure 2.7: Cycles needed for instruction dispatch. Other costs vary with the dispatch method (see text).

```
typedef void *Inst;

void engine()
{
  static Inst program[] = { &&add /* ... */ };
  Inst *ip;
  int *sp;

  goto *ip++;

 add:
  sp[1]=sp[0]+sp[1];
  sp++;
  goto *ip++;
}
```

Figure 2.8: Direct threading using GNU C's "labels as values"

Fig. 2.8). If portability to machines without `gcc` is a concern, it is easy to switch between direct threading and ANSI C conforming methods by using conditional compilation.

If the instructions are of constant length, dispatching the next instruction can be performed in parallel with the processing of the current instruction. This is very useful for filling delay slots of both the instruction dispatch routine and the rest of the instruction. When coding in C, care must be taken to avoid potential dependences due to aliasing (e.g., between instruction and stack pointer) that would prevent the compiler from performing good scheduling. In other words, in the C code the load of the next instruction must come before any stores of instruction results (e.g., into the stack). If an even higher amount of instruction-level parallelism is desired, a part of the dispatch routine (e.g., instruction fetch) can be shifted to earlier instructions. However, this work is wasted if the virtual machine control flow changes (unless there are delayed branches in the virtual machine).

2.1.2 Semantic content

The interpreter overhead can also be reduced by reducing the number of primitives executed, i.e., by increasing the semantic content of each instruction. Combining often-used instruction sequences into one instruction is a popular technique, as well as specializing an instruction for a frequent constant argument (eliminating the argument fetch and enabling optimizations in the native code for the instruction). Care has to be taken that the resulting code expansion with its higher real machine instruction cache miss-rate does not cancel out the benefits. Also, often the compiler must be made more complex to make use of these instructions. On the other hand, optimizing compilers can make instructions with high semantic content useless (part of the RISC lesson).

[Pro95] used tree pattern matching to find and combine the most frequently used tree patterns in *lcc*'s intermediate language [FH91a, FH95]. He then compiled the intermediate code into a stack-based byte-code. He achieved speedups of 2–3 by adding 10–20 instructions (*superoperator*s). This high speedup is probably due to the fact that *lcc*'s intermediate language was not designed for being interpreted.

Linear peephole optimization of Forth code can reduce the number of instruction dispatches by (a very conservative) 22% by adding 100 instructions (see http://www.complang.tuwien.ac.at/forth/peep/). Peephole optimization of small benchmarks has produced speedups of about 50% (see Section 4.5).

2.2 Accessing arguments

2.2.1 Virtual machine architecture

In the hardware area, the contest between stack and register architectures has been decided for register machines.[2] However, for interpretive implementations the picture looks different:

From the view of the compiler writer, many languages can be easily compiled for stack machine code. To achieve better performance with a register machine, the compiler must perform optimizations, e.g., global register allocation (which needs data flow analysis). This would eliminate one of the advantages of using an interpreter, namely simplicity.

Moreover, in an interpreter the spill[3] and move instructions necessary in register architectures are much more time consuming than in hardware, since

[2] For a dissenting opinion, read [Koo89].

[3] If there are more values than the compiler can keep in registers, some values have to

```
lw   $3,0($6) #get register numbers,
lw   $2,4($6) #$6=instruction pointer
lw   $4,8($6)
addu $3,$7,$3 #add reg. array base ($7)
addu $2,$7,$2
lw   $2,0($2) #load arguments
lw   $3,0($3)
addu $4,$7,$4
addu $2,$2,$3 #perform operation
sw   $2,0($4) #store result
```

Figure 2.9: Add in a register architecture (without instruction dispatch)

```
addu $5,$4,$6  #$5=r3 $4=r1 $6=r2
```

Figure 2.10: Unfolded add (r1 and r2 into r3)

each instruction also has to execute an instruction dispatch. This is not balanced by the fact that the other instructions also have to perform instruction dispatches, since the other instructions usually have higher semantic content. I.e., the proportion of spill code is higher for virtual register machines than for real register machines.

In hardware, the instruction and the register numbers are decoded in parallel. A simple software implementation of a register machine has to fetch and/or decode the register numbers using separate instructions. Even with the amount of instruction-level parallelism that superpipelined and superscalar processors offer today and in the near future, this still costs much time. Since hardware registers cannot be accessed in an indexed way, the virtual machines registers have to be kept and accessed in memory, costing even more time. Fig. 2.9 shows a three register add (without instruction dispatch) on the MIPS architecture (10 cycles on the R3000).

There is an alternative implementation of a register machine: The registers accessed can be encoded into the instruction by unfolding it, i.e., by creating a version of the instruction for every combination of registers. The instruction can access the registers directly, and therefore the registers can reside in real machine registers, if there are enough[4]. Fig. 2.10 shows one

be stored into memory and loaded back later. This is called *spilling*.

[4] However, the availability of registers should not be taken for granted even on register-rich RISCs. E.g., when I tried to keep the top of stack (of Forth's stack-oriented virtual machine) in a register on the MIPS architecture, `gcc` (versions 2.3.3 and 2.4.5) spilled the return stack pointer to memory, an important internal register of the virtual machine.

```
lw   $2,0($5) #get arguments
lw   $3,4($5) #$5=stack pointer
addu $5,$5,4  #update stack pointer
addu $2,$2,$3 #perform operation
sw   $2,0($5) #store result
```

Figure 2.11: Add in a simple stack implementation

version of the add instruction. However, this strategy causes code explosion, and will probably suffer a severe performance hit on machines with small first-level caches: E.g., there would be 288–512 versions of every three-register instruction in a virtual machine with 8 registers (the lower bound is for commutative operations); the add instruction alone would need 4.5 KB in a direct threaded implementation on the MIPS architecture. The size of the first-level (real machine) instruction cache on the R4000 is just 8 KB.

A simple stack machine does better than a simple register machine (see Fig. 2.11). It has the same number of operand fetches and stores; in addition, many instructions update the stack pointer. But there is no fetching/decoding to learn where the operands are.

A simple method to improve stack machine interpreter performance is to keep the top-of-stack in a register (see Fig. 2.13, Section 2.3.1).

2.3 Stack caching

If there are enough registers, the number of operand fetches and stores can be reduced by keeping stack items in registers. I call this *stack caching*, in analogy to other methods to reduce access overhead by keeping frequently-accessed data in high-speed storage.

2.3.1 One-state stack caches

The simplest approach keeps a fixed number of stack items in registers (see Fig. 2.12 and 2.13). In contrast to the approaches that I will discuss later, here the stack cache has only one state, so there is no need to keep track of the state; consequently, there is no need to change the interpreter dispatch method or the compiler to use this method.

How many loads and stores does this method eliminate? How many registers are useful for keeping stack items? Can it hurt to keep more stack items in registers?

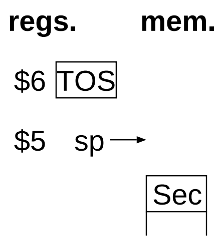

Figure 2.12: A stack implementation, where the TOS is in a register

```
lw   $2,4($5) #get other argument, $5=sp
addu $5,$5,4  #update stack pointer
addu $6,$6,$2 #perform operation, $6=tos
```

Figure 2.13: Add, the top of stack is kept in a register

Let us assume that n items are kept in a register. Now, consider an instruction that takes x items from the stack and stores y items to the stack (i.e., it has the stack effect $\mathsf{v}_1\ldots\mathsf{v}_x$ -- $\mathsf{w}_1\ldots\mathsf{w}_y$)[5]. We have to consider a few cases:

$x = y$ The instruction loads and stores only $x - n$ items. So, the larger n, the better (until $n \geq x$).

$x \geq n \wedge y \geq n$ The instruction loads $x - n$ and stores $y - n$ items. Again, the larger n, the better.

$x \geq n \wedge y < n$ The instruction loads $x - n$ items as operands and $n - y$ items to keep the cache full. If we assume that all loads cost the same, changing n has no effect as long as it does not change the precondition. However, loading operands may be more expensive, due to pipeline stalls; in that case, a larger n would be better.

$x < n \wedge y \geq n$ Symmetric to $x \geq n \wedge y < n$.

$x \neq y \wedge x < n \wedge y < n$ The instruction loads or stores $|x - y|$ stack items to keep the cache full; it moves $n - \max(x, y)$ otherwise unaffected stack items between registers within the cache. So, the smaller n, the better.

[5] Some stack manipulation instructions do not use the stack items they consume, so there is no need to load the stack item; others put items that they fetch back in the same place one the stack, so there is no need for the store. In the present discussion we ignore these special cases and assume that all vs have to be loaded and all ws have to be stored.

So the answer to the questions posed above depends on how often the different kinds of instructions are used and, to some degree, on the characteristics of the real machine. If we assume that all loads cost the same no matter where they appear (and likewise with stores), then we get the following rules, that hold if loads, stores, and moves cost more than zero cycles. Keeping the top n items in registers

- is better than keeping just $n-1$ items, if $x \geq n \ \wedge \ y \geq n$, due to fewer loads from and stores to the stack.
- is usually slower than keeping $n-1$ items, if $x \neq y \ \wedge \ x < n \ \wedge \ y < n$, due to additional moves between registers.
- is as fast as keeping $n-1$ items in the other cases.

Keeping one item in a register is never a disadvantage (if there are enough registers). Whether keeping two items is a good idea, depends on the virtual machine and how it is used. E.g., for Forth it is not a good idea (see Section 2.6).

If we take instruction-level parallelism (i.e., nonuniform costs for loads and stores) into account, the optimum number of registers according to the rules above is the lower bound. I.e., machines that can exploit a high amount of instruction-level parallelism may profit from the prefetching effect of keeping more items in registers. On a related note, keeping one item in a register is especially useful for floating-point and other long-latency instructions, where the store back to the stack would expose the latency.

2.3.2 Multi-state stack caches

Keeping a constant number of items in registers is simple, but causes unnecessary operand loads and stores. E.g., an instruction taking one item from the stack and producing no item (e.g., a conditional branch) has to load an item from the stack, that will not be used if the next instruction pushes a value on the stack (e.g., a literal). It would be better to keep a varying number of items in registers, on an on-demand basis, like a hardware cache.

This means that the cache can be in several states. Every allowed mapping of stack items to machine registers constitutes a *cache state*. This requires different implementations of an instruction for different cache states.

There are several sensible options on the set of states allowed. Basically, we would like the set to be finite, so we can use finite state machines to describe the effect of executing or compiling instructions. The relations of the states should minimize the amount of work necessary for getting from

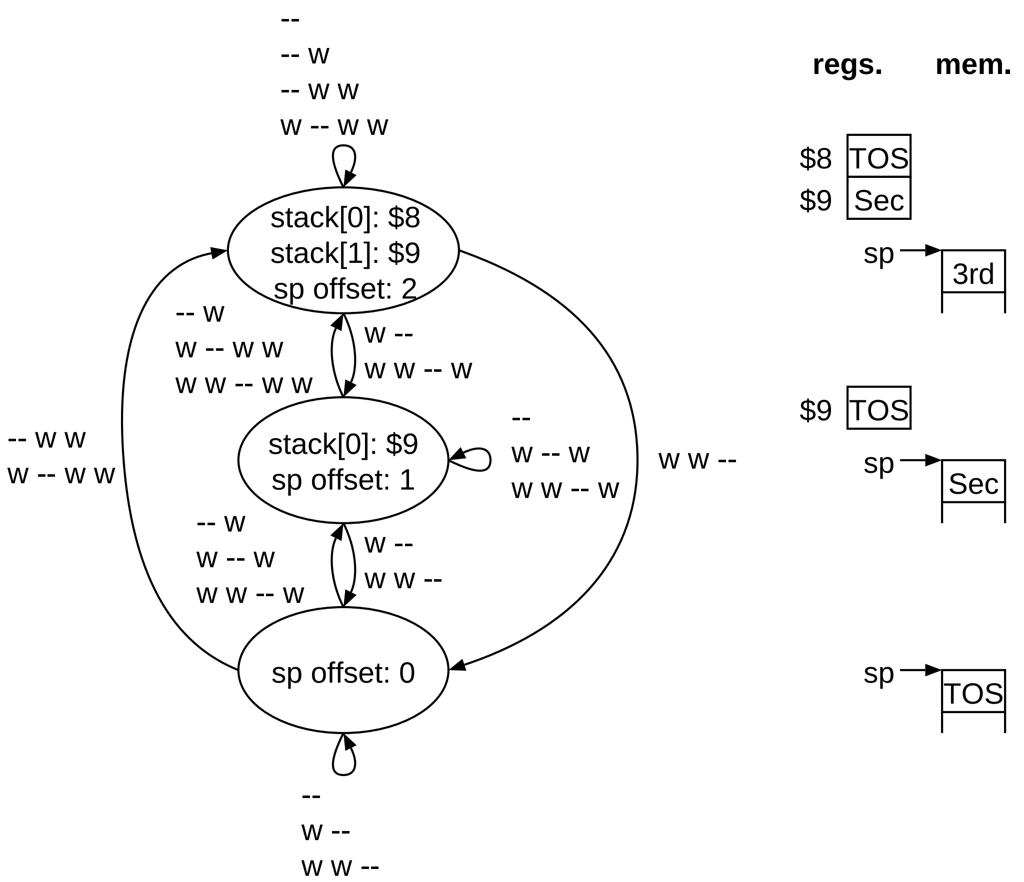

Figure 2.14: A simple cache state machine. Transitions are marked with stack effects; all instructions with the same stack effect perform the same transitions. The three states of the stack cache are also shown in a style similar to earlier figures.

one state to another. Fig. 2.14 shows a three-state machine for stack caching in two registers. Transitions are shown for words with various stack effects (due to space limitations not for all stack effects).

In general, the selection of a set of states and transitions for a given number of states and registers is an optimization problem that we leave for future work. Here we present just a few insights.

Stack pointer updates

In addition to stack accesses, many stack pointer updates can be optimized away, too: The cache state can also contain the information how much the contents of the stack pointer register differ from the actual value of the stack

```
addu $9,$8,$9
```

Figure 2.15: Add in stack caching (starting in the full state of the three-state machine)

pointer. A good strategy that does not introduce additional states is to let the difference correspond to the number of stack items in the cache (see Fig. 2.14). This means that the stack pointer need not be updated in instruction implementations that can access all stack items in registers, i.e., hopefully most of the time.

Stack caching with stack pointer update minimization leads to code that is as good as that of the unfolded register machine (see Fig. 2.15).

Minimal organization

As a minimum, there should be one state for every number of stack items in registers (as in Fig. 2.14). To minimize the amount of work, the bottom of the cached stack items should be in the same register in all states; the other stack items should be allocated likewise. This arrangement of states avoids the need to move stack items around on the bottom of the cache whenever something on the top changes.

Overflows and underflows

There is a movement cost, however: If something has to be pushed when the cache is full, all stack items in the cache have to be moved to other registers. Fortunately, overflows are very rare if the cache is sufficiently large (if the cache is small, there are not many moves). It can be made rarer by choosing an appropriate followup state for overflowing instructions:

Choosing the full state as overflow followup state minimizes the traffic between the stack cache and memory, but not the number of moves and stack pointer updates. In particular, on processors where a move costs the same as a store, the transition to any state costs the same. So it can be better to choose a non-full state as the overflow follow-up state (see Fig. 2.16), in order to reduce the number of overflows (even though this increases the number of underflows a bit). Which state is the best, should be determined empirically (see also Section 1.4.1).

In the same way an optimal followup state for underflow can be selected. It is probably useful to put at least the values in registers that the underflowing instruction produces, i.e., the underflow followup state depends on the executed instruction.

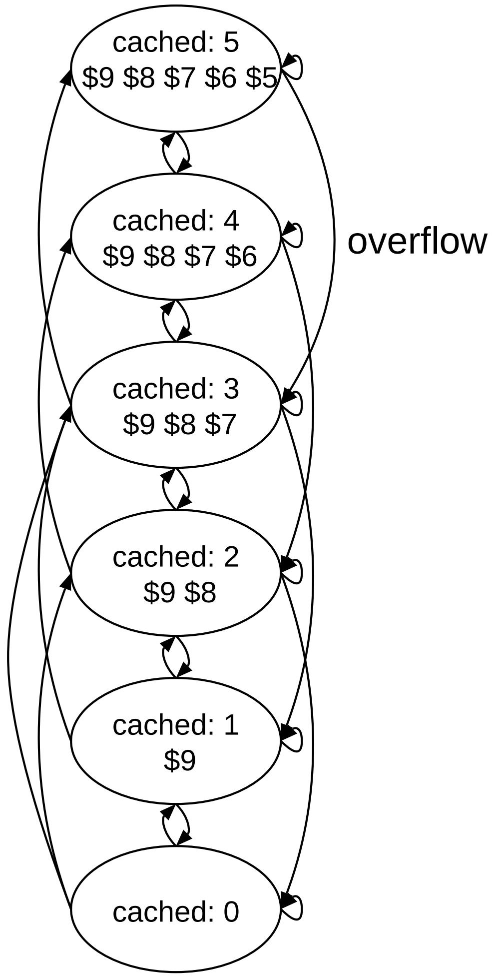

Figure 2.16: Overflow transition in a minimal organization (the top-of-stack is rightmost, `$9` contains the deepest cached item)

Note that the optimal followup states for overflow and underflow depend on each other. I.e., if a suboptimal behaviour for underflows is selected, the corresponding overflow followup state will tend to be fuller than for the optimal underflow behaviour, in order to reduce the number of costly underflows.

Another solution to the movement problem on overflows is to introduce more states: instead of moving all stack items, just the bottom cached stack item is stored to memory and the register where it resided is reused to keep the top of stack. Of course, this new mapping of stack items to registers has to be represented in a new state. The moves would have to be performed when the new state is left. To avoid this, appropriate neighbours for this new state should be introduced. If this approach is performed consequently, all

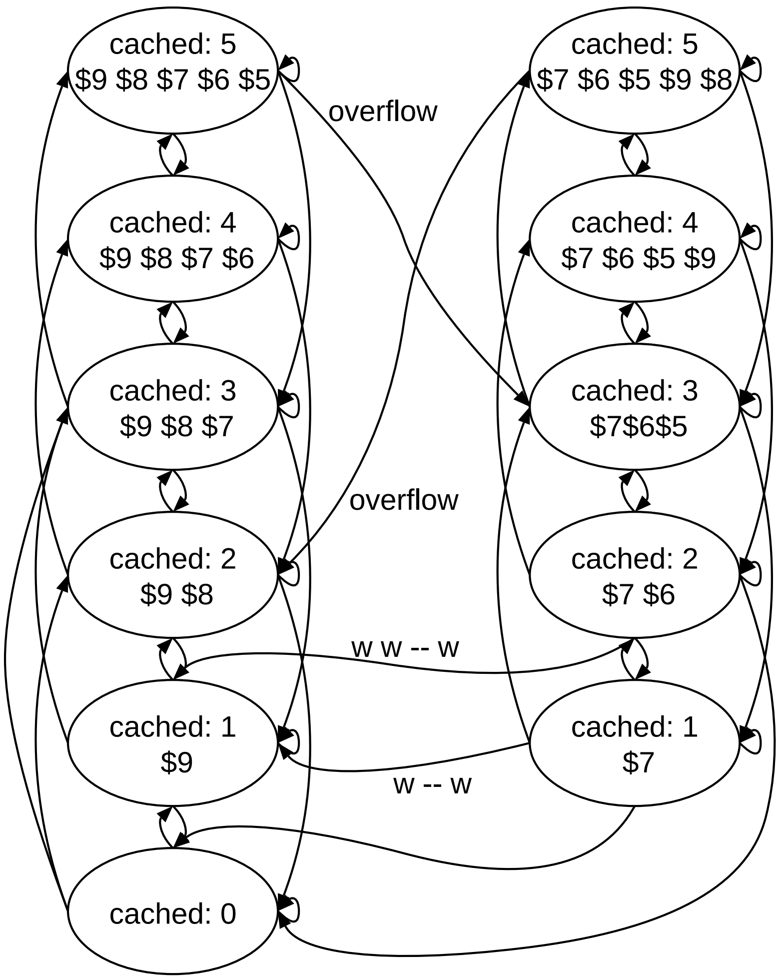

Figure 2.17: Avoiding moves with additional states

such moves can be eliminated, but the number of states is nearly multiplied by the number of cache registers. Combinations of both solutions to this problem are possible (see Fig. 2.17).

Stack manipulation instructions

Stack manipulation instructions also cause moves in the minimal state machine. As before, these moves can be optimized away by introducing more states. For stack shuffling instructions (e.g., `swap` and `rot`), the extreme form of this approach creates all assignments of cached stack items to cache registers where no register occurs twice. To optimize duplicating instructions (e.g., `dup` and `over`), we can introduce states where one register holds

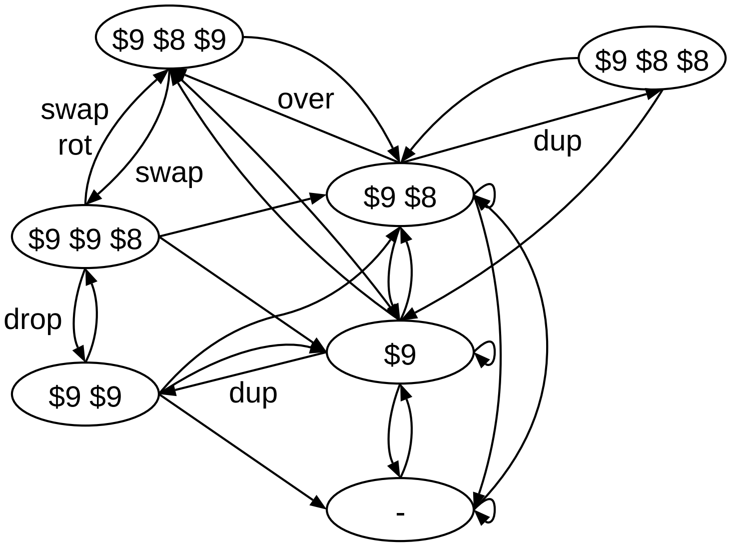

Figure 2.18: A cache organization where one duplication is allowed (`dup` (x -- x x) duplicates the top of stack, `over` (x y -- x y x) duplicates the second item, `swap` (x y -- y x) swaps the two top items, `rot` (x y z -- y z x) rotates the third item to the top, `drop` (x --) pops the top item)

several stack items (that have the same value). In extreme form, this results in an infinite number of cache states, because an unlimited number of such instructions causes an equally unlimited number of stack items to reside in the cache, and an infinite number of states is needed to record all these possibilities. If the number of cache states is to be limited, the number of duplications represented in the states has to be limited. E.g., the number of stack items in the cache could be limited, the number of duplicates of each item, or the total number of duplications. Figure 2.18 shows a two register cache organization where one duplication is allowed.

Multiple stacks

If there are several stacks, the simple solution is to treat them separately, with separate caches (and separate state machines). They can also be treated in a unified manner, sharing the same set of registers. Moves between the stacks can be optimized away by introducing additional states.

In Forth, return stack caching would be profitable, given the high frequency of calls and returns. A nice optimization is possible here: The (virtual machine) instruction pointer (IP) need not be moved to the top of return stack register during the call, instead the register containing the old IP can be treated as the top of return stack register and the new IP resides in another

registers	1	2	3	4	5	6	n
"minimal"	2	3	4	5	6	7	$n+1$
overflow move opt.	2	5	10	17	26	37	n^2+1
arbitrary shuffles	2	5	16	65	326	1,957	$\sum_{i=0}^{n} n!/i!$
$n+1$ stack items	3	15	121	1,356	19,531	335,923	$\sum_{i=0}^{n+1} n^i$
one duplication	3	7	14	25	41	63	$n(n+1)(n+2)/6+n+1$
two stacks	3	6	9	12	15	18	$3n$

Figure 2.19: The number of cache states

register.

Reducing the number of states

In practice finiteness is not enough, there are also other limits to the number of states. Figure 2.19 gives an idea of the number of states of various cache organizations with a varying number of registers. The "minimal" organization has only one state for a certain number of stack items in registers; "overflow move optimization" removes the moves on overflow by introducing more states while overflowing to a full state; "arbitrary shuffles" has states that allow the elimination of the moves of shuffle instructions; "$n+1$ stack items" supports keeping up to $n+1$ stack items in n registers, in any order and with any kind of duplication; these two cases show that the number of states can grow explosively. "One duplication" is the "minimal" organization, extended with states that represent one (arbitrary) duplication of a stack item. "Two stacks" is the "minimal" organization, combined with caching up to two items of another stack in the same registers, also in a "minimal" organization.

For organizations with many states, nearly all states will be rarely used. If a smaller number of states is desired, many of these states can be eliminated. Transitions to such states have to be rerouted, possibly incurring higher transition costs. However, these costs have to be payed rarely, only when the state would have been used.

This brings up the question of what transitions there should be in the first place. The simplest criterion is the cost of the transition itself. However, there are often several transitions costing the same (e.g., consider the overflow case in the "minimal" organization). In such cases a transition should be chosen to the node that has the smallest average transition cost (e.g., the state in the above-mentioned overflow case, that minimizes the costly overflows and underflows). Indeed, the cost of the transition should be considered to include the average transition cost of the successor node.[6] Or, even better, if the future is known (as in static stack caching, see Section 2.5), the actual future cost can be used to select the transition.

The choice of transitions also influences the usage counts of the states. It is desirable to have a strongly biased distribution of usage counts, in order to be able to eliminate many states, but also to achieve high real machine instruction cache hit rates. This biasing can be achieved by selecting a specific state and choosing transitions that get closer to this canonical state if there is a choice.

Prefetching

If stack item prefetching is desired, states with too few stack items in registers should be forbidden. This will cause slightly higher memory traffic: the prefetches will be useless if a number of pushes follows that causes the stack cache to overflow. In addition, on overflow the prefetched values have to be stored into memory, unless the cache state also contains information about the prefetched values (corresponding to *dirty* bits in hardware caches). Prefetching more than one value can also introduce moves (an underflow variant of the overflow problem). If it is used, prefetching should overcompensate these costs by reducing the number of delay slots.

2.4 Dynamic stack caching

In dynamic stack caching the interpreter maintains the state of the cache and the compiler need not even be aware of the existence of a stack cache. This means that there is a copy of the whole interpreter for every cache state. The execution of an instruction can change the state of the cache, and the next instruction has to be executed in the copy of the interpreter corresponding to the new state.

This implies a change of the instruction dispatch routine. In a switch-based implementation, the instruction just has to jump to the appropriate copy of the switch. For direct threading the changes are not so simple: The easy solution performs a table lookup (see Fig. 2.20). This costs a (real machine) load instruction on current RISC processors; to make bad news worse, this load instruction may cost more than one cycle, since it increases the data dependence path length of the instruction dispatch sequence, which will often become the critical path of an instruction, especially if much of the

[6]This infinitely recursive definition would result in infinite costs, but it is possible to shift the scale into a finite range.

```
$L2: #add in state 0: cache empty
 lw   $4,0($6) #get arguments,
 lw   $3,4($6) #$6=stack pointer
 lw   $2,0($5) #get next instruction, $5=instp
 addu $6,$6,8  #stack pointer update
 lw   $2,4($2) #table lookup, next state: 1
 addu $5,$5,4  #advance instruction pointer
 j    $2       #jump to next instruction
 addu $4,$4,$3 #operation

$L3: #add in state 1: tos in $4
 lw   $2,0($6) #get other argument
 lw   $3,0($5)
 addu $6,$6,4  #stack pointer update
 lw   $3,4($3) #next state: 1
 addu $5,$5,4
 j    $3
 addu $4,$4,$2 #operation

$L4: #add in state 2: tos in $7, second in $4
 lw   $2,0($5)
 #nop
 lw   $2,4($2) #next state: 1
 addu $4,$4,$7 #operation
 j    $2
 addu $5,$5,4
```

Figure 2.20: Add in dynamic stack caching with table lookup

rest has been optimized away (as in the add in state 2 in Fig. 2.20). On CISCs the lookup may come for free (i486) or at little cost. The other solution is to store the instructions for a state at a fixed offset from the corresponding routines in the other states. Then the address of the routine for an instruction can be computed by adding the base address of the instruction and the offset of the state. This costs a (real machine) add instruction on many processors, but may come for free on others (SPARC). The problem with this approach is that no portable language I know supports placing routines at specific points in memory; what's worse, even some assemblers do not support it (e.g., the MIPS/Ultrix assembler).

If instruction dispatch becomes more expensive, dynamic stack caching is probably not worth the trouble. E.g., none of the add implementations in Fig. 2.20 is faster (on both the R3000 and R4000) than the add in Fig. 2.13

input code		`dup`			`1+`		
cache state	`$9`	$\to$	`$9`	`$9`	$\to$	`$9`	`$8`
generated code				`1+`$_{\$9\to\$8}$			

Figure 2.21: Compilation of `dup 1+` for static stack caching with the cache organization of Fig. 2.18. `1+` (`n1` -- `n2`) increments the top-of-stack.

with direct threading.

Since the whole interpreter has to be replicated for every state, only state machines with a few dozen states or less are practicable (depending on the size of the interpreter and the (real machine) instruction cache). In other words, the stack cache should have the minimal organization, maybe with a few frills like a bit of return stack caching, or, if there are few registers for caching, one duplication, to make better use of them. In many cases eliminating the moves of stack manipulation instructions does not pay: The instruction dispatch has to be performed anyway, and the moves can often be done in parallel, i.e., in the delay slots.

Since the state of the cache is represented in only one value, i.e., the program counter of the processor, it is not possible to treat two caches (e.g., for an integer and a floating-point stack) with separate state machines in dynamic caching. The states of both caches have to be represented in a single state machine. This multiplies their number and makes big caches for more than one stack impractical.

2.5 Static stack caching

In static stack caching the compiler keeps track of the state of the cache and generates the code accordingly (see Fig. 2.21). Of course, we now have to distinguish between the stack-based virtual machine code that the higher level of the compiler sees, and the code that is generated and interpreted, where the stack cache is visible.

Static stack caching offers several big advantages over dynamic stack caching:

- There is no need for a special instruction dispatch routine and its possible performance disadvantages; we can use direct threading.

- Stack manipulation instructions can be optimized away completely, i.e., not even an instruction dispatch is executed. The compiler just notes the state transition (see Fig. 2.21).

- There is no need to replicate the whole interpreter for every state: The implementation of the same instruction in many states can be the same, e.g., when the arguments of the instruction are accessed in the same registers, but some other stack items reside in different registers (in dynamic stack caching they would have different instruction dispatch routines for continuing in different states). Moreover, implementations of rarely used instructions for rarely used states can be left out. The compiler will then insert code for a transition into a state for which the instruction is implemented.

  Caches with several thousand states are feasible, and probably even more with table compression techniques.

- The compiler knows the future instruction stream and can generate optimal code for it.

Of course, there is also a disadvantage: It is not possible to execute the same code in different states. The compiler has to reconcile the states of different control flows at control flow joins. Apart from this fundamental problem there are also the practical problems of insufficient knowledge in the compiler and avoiding compiler complexity. In particular, the compiler usually knows nothing about the states of callers and callees.

The traditional solution for the call problem is to have a calling convention. In the case of stack caching this means that all procedures start in a specific state and return in a specific (possibly different) state. The transition into these states can be performed by the call and return instructions respectively.

A simple solution for the control flow join problem is to have a "control flow convention": at every basic block boundary (i.e., at every branch and branch target) the code is in a canonical state. The transition into this state can be performed by the branch instructions. For branch targets the transitions have to be performed by the instruction before the target. A slightly more complex, but faster solution is to have the branch perform the transition to the state at the branch target without causing a reset to a canonical state before the branch target.

Due to the need for a calling convention, a return stack cache cannot be used as effectively as in dynamic stack caching. However, a one-register return stack cache can be used to good effect: at the start of a procedure the register is filled with the return address. This is equivalent to the leaf procedure optimization on RISCs.

Generating optimal code using knowledge of the next instructions in the basic block is possible in linear time using a two-pass algorithm, as a spe-

Prog.	Instr.	loads	updates	rloads	rupdates	calls	CPI
compile	1,562,172	0.76	0.55	0.17	0.32	0.13	
gray	1,588,545	0.69	0.43	0.21	0.39	0.17	
prims2x	5,766,854	0.75	0.43	0.18	0.34	0.16	10
cross	4,914,610	0.74	0.51	0.19	0.33	0.14	14

Figure 2.22: The measured programs and some of their characteristics: instructions, loads from (=stores to) the stack, stack pointer updates, return stack loads/stores, return stack pointer updates, calls per instruction and cycles per instruction (on a DecStation 3100).

cialization of the approach taken in tree pattern matching [PLG88, FHP91]. The first pass just determines which of the possible code sequences is optimal, the second pass then generates the code. Both passes use finite state machines and are therefore fast. The usefulness of this technique depends on the organization of the cache state machine. It is only useful if there is more than one transition possible for an instruction from a given state and if choosing the right one requires foresight.

From a certain point of view there is not much difference between static stack caching and using a register architecture for the virtual machine. Indeed, it can be seen as a framework to make virtual register machines more usable: It provides automatic register allocation and spilling without lots of overhead instructions. It also provides principles for keeping the number of different implementations of an instruction small, if necessary. And it provides a simple, stack-based interface to the higher levels of the compiler. And the low level of the compiler does not have to handle the complexities of register allocation, it is just a simple and fast state machine. However, there is quite a bit of complexity in the generator that generates the instructions and the tables for the compiler.

2.6 Empirical results

I instrumented a Forth system to collect data about the behaviour of various stack caching organizations.[7] Several real-world applications were used as benchmarks: interpreting/compiling a 1800-line program (*compile*), running a parser generator on an Oberon grammar (*gray*), a text filter for generating C code from a specification of Forth primitives (*prims2x*), and a cross-compiler generating a Forth image for a computer with different byte-order

[7] The raw data is available at ftp://ftp.complang.tuwien.ac.at/pub/misc/stack-caching-data.

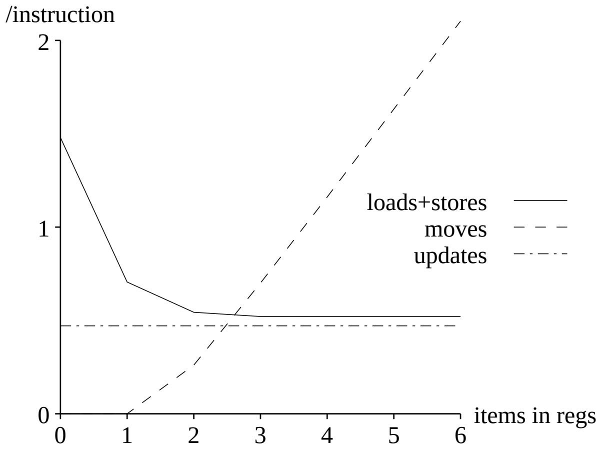

Figure 2.23: Keeping a constant number of items in registers: Memory accesses, moves, and stack pointer updates per instruction vs. number of items kept in registers

(*cross*). Figure 2.22 shows the number of loads from the stack (same as the number of stores to the stack), stack pointer updates and executed (virtual) instructions for these programs on an implementation without any kind of stack caching. It also gives the number of return stack loads/stores and stack pointer updates. The return stack is not considered in the rest of the measurements. Applying return stack caching should have similar effects as for the data stack, with one exception: Most return stack accesses are simple pushes (on calls, `-- w`) or pops (on returns, `w --`); therefore, always keeping one return stack item in a register has hardly an effect (see Section 2.3.1).

To compare the total argument access overhead of various organizations, the components have to be weighed and added. We used the following weights: loads, stores, moves and stack pointer updates cost one cycle, instruction dispatches cost four cycles. Since the number of loads from and stores to the stack in memory is equal, we will only display their sum in the figures. The figures display the total sums for all programs.

First, we measured the effect of keeping a **constant number of stack items** in registers (see Fig. 2.23). It is easy to see that keeping one item in a register is best (see also Fig. 2.28): It significantly reduces the number of loads and stores. Keeping more items in registers reduces loads and stores, but introduces too many moves to be useful. Of course, the number of stack pointer updates cannot be reduced with this technique. On a DecStation 3100 (16MHz MIPS R2000[8]) keeping one item in a register causes a speedup of 11% for *prims2x* and 7% for *cross*.[9] This is a little better than what I ex-

[8]The R2000 has the same instruction timings as the R3000.

[9]The other programs run too fast to produce exact timings. Explicit register declara-

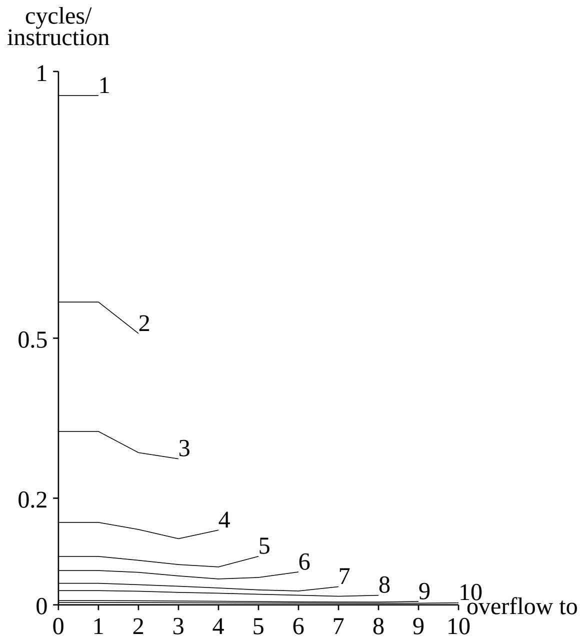

Figure 2.24: Dynamic Stack Caching: Argument access overhead in cycles/instruction of minimal organizations with different numbers of registers vs. overflow followup state

pected based on the reductions in loads and stores (8% and 5%, respectively); apparently these loads and stores cost more than one cycle.

Next, we measured **dynamic stack caching** on minimal organizations with a varying number of registers and varying overflow followup states. We did not optimize the underflow followup state; instead, we used the state that has those items in registers that the underflowing instruction produces. In Fig. 2.24 the lines represent the performance of cache organizations that use a specific number of registers, while varying the overflow followup state. E.g., the line labelled with "4" represents the organizations that use four registers; the lowest (optimal) point on this line is for the organization that uses state 3 as the overflow followup state, i.e., the state where the registers contain the top three stack items and one register is free. The argument access overhead is approximately halved for every register that is added.[10]

Figure 2.25 shows how the components of the argument access overhead vary for different overflow followup states of organizations with six registers. The fuller the overflow followup state, the more overflows there are, increasing the number of moves. At the same time, the memory traffic decreases, since less data, that would have fit into the cache, is stored and later loaded again.

tions were used to keep `gcc` from spilling important registers.

[10]This holds up to about 14 registers, then the decrease slows down.

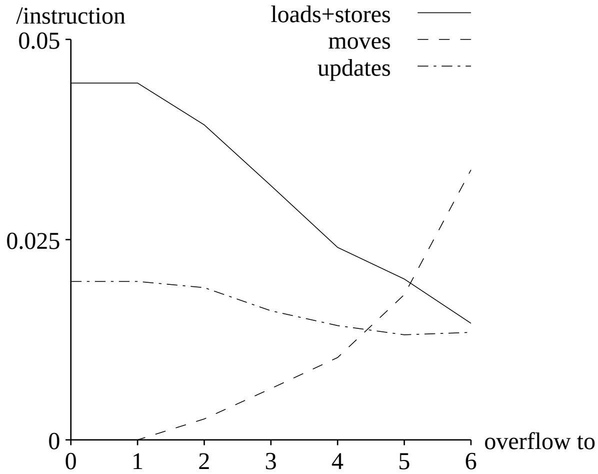

Figure 2.25: Dynamic Stack Caching: Memory accesses, moves, and stack pointer updates per instruction of organizations with 6 registers for varying overflow followup states

Although the number of overflows increases, the number of stack pointer updates decreases, because the increase in overflows is outweighed by the decrease in underflows. My explanation for this observation: There is a strong tendency to return to an earlier stack depth without going any deeper in the meantime (see Section 1.4.1) and underflows are usually one item at a time, while overflows to-non-full states spill several items at a time (for higher number of registers the number of stack pointer updates is not minimal when overflowing to the full state).

For **static stack caching** I looked at organizations based on minimal organizations, that also contained states that represent the application of one stack manipulation word to a state of the minimal organization (but only if the arguments of the stack manipulation word are already in registers). These organizations were combined with the control flow convention approach; I tried all the states of the minimal organization as canonical state (which also served as overflow followup state). In Fig. 2.26 the lines represent the performance of cache organizations that use a specific number of registers, while varying the canonical state. I.e., the line labelled “4” represents the organizations that use four registers. The lowest point on this line is for using state 2 as the canonical state, i.e., the state where two stack items are cached in two of the registers.

Due to the high frequency of cache resets to the canonical state, the best canonical state (for organizations with more than three registers) is the two-register state. It decreases the number of underflows fairly well without introducing too many moves on cache reset (see Fig. 2.27). Increasing the

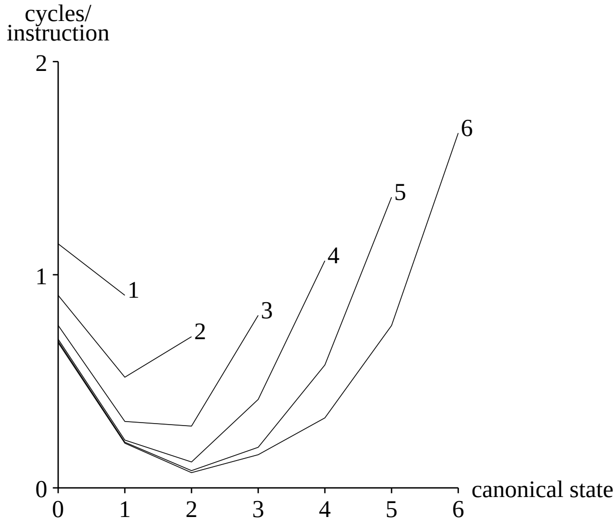

Figure 2.26: Static Stack Caching: Argument access overhead in cycles per (original) instruction of various organizations vs. canonical state

number of registers beyond five has hardly any effect, because the cache is usually reset before it overflows five registers. So the number of loads, stores, moves, and updates stays at a certain level (about 0.1 loads and stores and 0.2 moves and stack pointer updates per instruction), whereas it approaches 0 in dynamic caching. However, static caching also reduces the number of executed instructions. Note that Fig. 2.26 displays the overhead per original instruction; since instruction dispatch is not counted in the other figures, here the dispatches that are optimized away are subtracted from the other overhead.

The majority of cache resets in the programs I measured is caused by calls and returns. Indeed, in these programs every third or fourth instruction is a call or return. So, the best way to reduce the number of cache resets and to increase static stack caching performance in these programs would be procedure inlining. Note that a lower number of cache resets will increase the number of useful registers and change the optimal canonical state, asymptotically approaching the behaviour of dynamic stack caching.

I did not evaluate the effect of reducing the number of instances of the instructions. However, the distribution of the execution frequency of the instructions (10% account for 90% of the executed instructions) suggests that vast reductions are possible without causing a significant amount of additional state transition code.

Comparison. Figure 2.28 compares the three approaches. For dynamic and static stack caching the best of the evaluated organizations for a specific

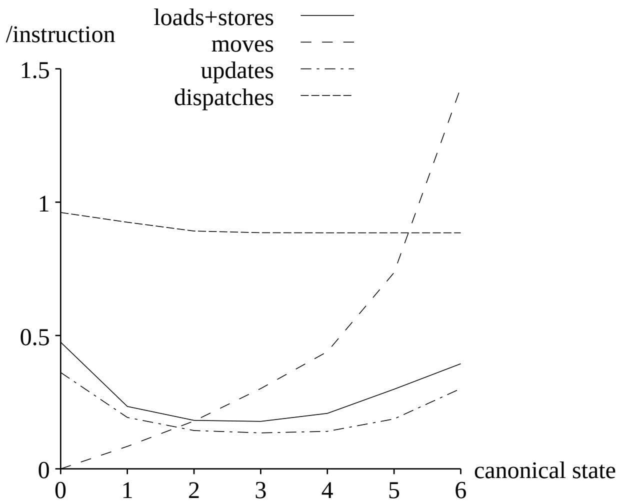

Figure 2.27: Static Stack Caching: Memory accesses, moves, stack pointer updates and instruction dispatches per (original) instruction of organizations with 6 registers for varying canonical states

number of registers was chosen. Note that the coincidence of the lines for dynamic and static stack caching is partly an artifact of the weights I have chosen for the various overheads, in particular, the weight of instruction dispatch. E.g., if instruction dispatch costs five cycles, static stack caching rivals dynamic stack caching also for higher numbers of registers; if instruction dispatch costs even more, static stack caching is better than dynamic stack caching everywhere, and its line would be partly below 0 (i.e., the dispatches optimized away outweigh the remaining argument access overhead). Note that this comparison does not consider the effects of the possibly higher instruction dispatch cost of dynamic stack-caching.

How much speedup can we expect from these improvements? That depends on how much time an instruction takes, on average. For *cross* and *prims2x* on a DecStation 3100, we can expect 17% and 25% speedup from static stack caching with 5 registers over a version that does not perform any stack caching (if we assume one cycle per real machine instruction, which has proven conservative). If we assume furthermore that half of the memory accesses and stack pointer updates for the return stack can be eliminated by having a one-register return-stack cache (i.e., bye performing leaf routine optimization), the speedup increases to 20% and 31%. Take these projections with a grain of salt. There are many issues involved that can influence the results, so a definite conclusion should only be drawn after timing a full implementation.

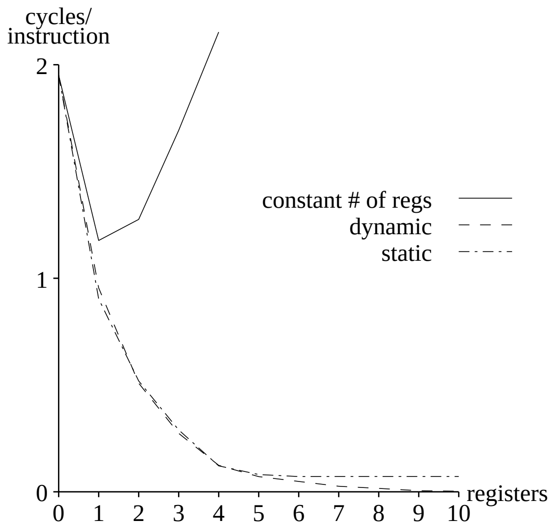

Figure 2.28: Comparison of the approaches: argument access overhead in cycles/instruction vs. number of registers used

2.7 Related work

Much of the knowledge about interpreters is folklore. The discussions in the Usenet newsgroup `comp.compilers` [`c.c`] contain much folk wisdom and personal experience reports.

Probably the most complete current treatment on interpreters is [DV90]. It also contains an extensive bibliography. Another book that contains several articles on interpreter efficiency is [Kra83]. Most of the published literature on interpreters concentrates on decoding speed [Bel73, Kli81], semantic content [Pro95], virtual machine design and time/space tradeoffs [Kli81, Pit87].

Stack caching has been used first in hardware stack machines [Bla77, HS85, HFWZ87, HL89, Koo89] and for speeding up procedure calls in processors designed at Bell Labs [DM82] and UC Berkeley (register windows) [HP90]. For interpreters, [DV90] proposed dynamic stack caching with a minimal cache organization without stack pointer update minimization and with the full state as followup state. They do not discuss other possible organizations and apparently they used only the Sieve benchmark for their empirical evaluation. They report speedups (probably over an implementation that does not keep any part of the stack in registers) of 16% for Forth on an 8086 with a two-register cache and 17% for M-Code (a virtual machine for Modula-2) on an 68020 with a three-register cache.

Chapter 3

Native-Code Compilation

Native code compilers offer flexibility and high execution speed, but they are also the most expensive implementation option (apart from special hardware), in particular, if the compiler should target several architectures.

Aggressive native code compilers can compile unconventional languages efficiently for mainstream (i.e., register architecture) hardware [Ung87, CU89, KB92]. Combined with the better manufacturing technology available to mainstream hardware this means that such compilers usually are the fastest (in terms of execution speed) implementations of the language.

This chapter discusses RAFTS, a framework for applying state of the art compiler technology to the compilation of stack-based languages, namely interprocedural register allocation (Section 3.2.4), inlining, tail call optimization (Section 3.2.3), instruction selection and scheduling (Section 3.2.1). Several new techniques are introduced that address the special needs of stack-based languages: methods for transforming programs into data flow graph (Section 3.2.1) and static single assignment form (Section 3.2.2), and treatment of unknown stack effects (Section 3.2.5) and dynamic stack manipulations (Section 3.2.7). Section 3.3 discusses a prototype implementation and Section 3.4 presents empirical results obtained with this implementation.

This chapter is a revised version of [Ert92]; Section 3.4 is based on [Pir95].

3.1 Related Work

BNF-based parsers can be used to transform postfix code into trees, i.e., data flow graphs. But parsers cannot process stack manipulation words and multiple stacks, they cannot produce DAGs and the method of Section 3.2.1 is simpler than parsing.

Stack-based languages have been used as intermediate code for compilers, e.g., [TvSKS83]. During code generation, these compilers face problems similar to Forth native code generation. However, they do not have to handle multiple stacks or stack manipulation words like `SWAP` and they do not use the stacks across basic block boundaries. Instead, they put a much higher emphasis on local and global variables. The code generator of the Amsterdam Compiler Kit (ACK) resembles the method of Section 3.2.1 in its use of the stack, but it performs all tasks of code generation at the same time [TvSKS83]. Everything else in ACK is different: In contrast to RAFTS, which does not do anything at the Forth level, ACK optimizes as much as possible at the postfix code level, because its main goal is machine-independence.

The minimization of stack pointer updates has already been discussed in [KKR65]: this paper describes an Algol 60 compiler that employs "simulation at compile time of the use of a conventional run-time stack" to reduce stack pointer updates. The run-time stack pointer is only updated at control-flow joins.

HP and Tandem have used binary translators to move legacy code base from stack architectures to (RISC) register architectures. [AS92] describes Tandem's binary translator, but gives few details about the stack-to-register conversion and is somewhat machine-specific in these details..

There are several Forth compilers that produce machine code [Ros86, Alm86, Pay91]. Usually they start by generating subroutine-threaded code, then they inline small subroutines, and peephole-optimize the resulting code to remove redundant pushes and pops. One nice property of such systems is that they can be easy to retarget. The retargetter just has to replace the native code for the primitives and the tables of the peephole optimizer; the compiler proper remains untouched. On these systems, much stack manipulation overhead remains. Rose [Ros86] reports that "the resulting machine code is a factor of two or three slower than the equivalent code written in machine language, due mainly to the use of the stack rather than registers"; our measurements confirm this (see Section 4.5).

However, some compilers are pretty sophisticated at reducing stack manipulation overhead. JForth V3.0 can keep up to five values in registers and optimizes words like `SWAP` and `ROT` completely away. This optimization is only performed on sequences of certain primitives. In contrast, RAFTS usually keeps stack items always in registers. Almy's non-interactive (i.e., batch) CFORTH compiler keeps up to two values in registers and reduces stack pointer manipulations [Alm86]. It keeps values in registers across basic block boundaries. However, the lack of global register allocation in his compiler would result in a lot of register shuffling at basic block boundaries, if the compiler used more than two registers. [Sto88, Kna93] show how a compiler using such techniques could be implemented: there is a compilation word for each word to be compiled that performs a case analysis of the preceding code; apart from being error prone, this method is hard to extend to global optimizations (e.g., global register allocation).

Koopman did similar research in the opposite direction: His C compiler keeps variables on the stack in order to execute C efficiently on stack machines [Koo92].

3.2 Compilation

I will start out with a very simplified Forth. Then I will add the features that complicate the compilation.

3.2.1 Basic Blocks

At the Forth level, basic blocks consist only of primitives like `+` and `!` (store), of literals, constants and variables, and of stack manipulation words like `SWAP` and `R@`. Words that change the control flow (e.g., `IF` or calls of high-level words) or are targets of control-flow changes (`BEGIN`, `THEN`) delimit basic blocks.

Data flow graphs

The data flow graph is the basic data structure of several state-of-the-art compiler algorithms[1], and it can be extended and converted to data structures for other algorithms with standard methods.

The compiler builds the data flow graph by symbolically executing the Forth words in the basic block: It maintains the stacks just as an interpreter would during execution. Stack manipulation words are compiled by simply executing them. When one of the other words is compiled, the compiler pops and pushes as many items as the word would during execution. But the word is not executed. Instead, the compiler builds a data flow graph node, which contains a token for the word (to indicate the kind of node) and the operands taken from the stack. A pointer to this data structure is pushed on the stack instead of the result. In this way, the compiler builds a data flow graph for the basic block (see Fig. 3.1).

There are a few complications:

[1] For basic blocks the graph is a directed acyclic graph (DAG); the data flow graph is so important that [ASU86] refers to it simply as DAG. Many compiler textbooks draw it up-side-down (just like they draw trees), with the data flowing upwards.

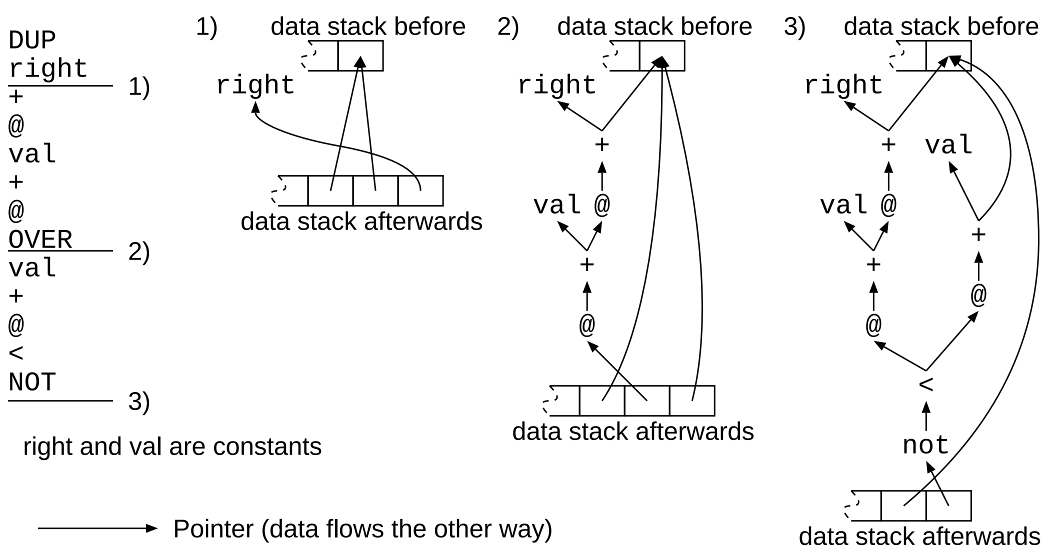

Figure 3.1: Building the data flow graph (three snapshots)

- Some words (e.g., `!`) push nothing on the stack, but they cannot be optimized away, because they have side effects in memory. So, these nodes cannot be found by performing a bottom-up graph-walk. The compiler has to keep track of these nodes in a separate list.

- The compiler must not use the regular data or floating-point stack for the symbolic execution, because the programmer may use them at compile time; the Forth standard specifies that words with normal compile-time semantics (like the words constituting basic blocks) have the compile-time stack effect `--` (i.e., no change). So the compiler must maintain separate stacks and perform the symbolic execution on them.

Code generation

The other three steps of the conversion of a basic block from Forth to native code are instruction selection, instruction scheduling and register allocation. They have been described extensively in the literature, so here they will not be explained in-depth.

Instruction selection combines the operators in the data flow graph into legal instructions of the target machine, transforming the operator DAG into an instruction DAG (see Fig. 3.2).

All referenced stack items now reside in pseudo-registers (px in Fig. 3.2),

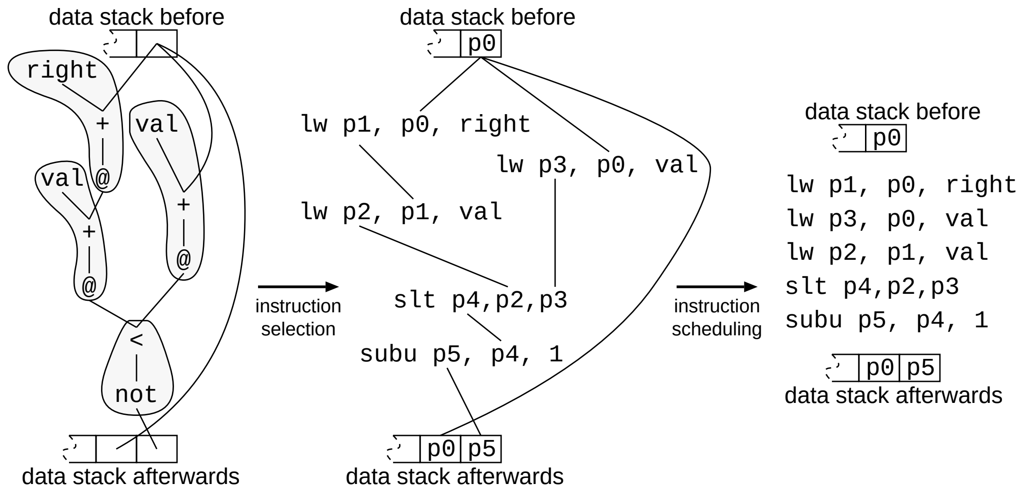

Figure 3.2: Instruction selection and scheduling (for the MIPS architecture)

of which an unlimited number can be used[2]; all stack accesses within a basic block have been eliminated. **Register allocation** replaces the pseudo-registers with real registers and spills excess pseudo-registers to memory. RAFTS uses interprocedural register allocation, which will be discussed in Section 3.2.4. In the meantime, the important thing to keep in mind is: Edges in the instruction DAG correspond directly to pseudo-registers; specifically, at basic block boundaries the pointers on the stacks correspond to pseudo-registers (already before instruction selection).

Instruction scheduling orders the instructions, i.e., it transforms the instruction DAG into a list (see Fig. 3.2). The data flow graph (as constructed above) contains only data flow dependences through the stack, it does not contain data dependences through memory; in other words, it is not a full data dependence graph. Therefore the compiler also keeps track of memory accesses during the symbolic execution: It remembers the last store and introduces dependences from the store to subsequent fetches and the next store; it also keeps a list of fetches since the last store and introduces dependences from the fetches to the next store.[3] Any topological ordering of the data dependence graph constitutes a valid schedule.

On RISCs the goal of instruction scheduling is to avoid pipeline stalls. The standard algorithm for RISC instruction scheduling is list scheduling

[2]Indeed, in order to make conversion to static single assignment form trivial, it is essential that no pseudo-register is reused.

[3]This method assumes that all memory accesses are aliased, which is safe, but may cost performance.

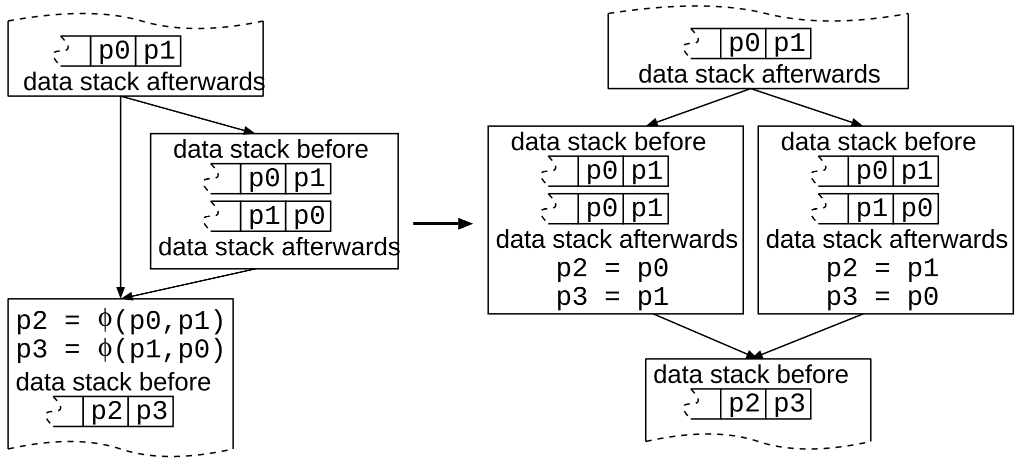

Figure 3.3: SSA form of "`IF SWAP ENDIF`" and conversion of ϕ-functions into moves

[LDSM80]. On CISCs instructions are scheduled to minimize register pressure. This is usually performed during local register allocation [ASU86, KPR91]. Instruction scheduling is usually performed before register allocation. Often, the instructions are reordered after register allocation to schedule the spill code (this requires building a new data dependence graph).

3.2.2 Control structures

Control structures connect basic blocks. This poses the problem of making adjacent basic blocks match.

Control flow splits (`IF`, and `UNTIL`) are easy: The values can stay in the registers they were in. Control flow joins (`THEN` and `BEGIN`) are a little harder[4]: The corresponding stack items of the joining basic blocks often do not reside in the same pseudo-register.

The compiler now inserts ϕ-functions after the join (see Fig. 3.3). The ϕ-functions do not correspond to real code (indeed, they are removed later); they indicate which values reach the join. The program is now in static single assignment (SSA) form [CFR$^+$91], which is an excellent representation for analysis and optimization purposes. Optimizations have been treated extensively in the literature and will not be discussed here.

After performing the optimizations, the SSA form can be converted into a more executable form easily: Every ϕ-function is replaced by moves in the parent basic blocks (see Fig. 3.3). If one of the joining basic blocks has

[4]Alternatively, we could choose to make joins easy and splits harder by stating that values stay in the same registers during joins.

several successors (i.e., it ends with a control flow split), the moves have to be inserted into a new, empty basic block. Many of the inserted moves will be removed by register allocation.

3.2.3 Calls

Words and their calls are treated like other control structures: A word's entry point is a control flow join, so a consistent state has to be reached. This is again represented by ϕ-functions at the entry point, this time with as many arguments as there are call sites. Later they will be converted into moves at the call sites. Like other control flow splits, the `EXIT`s pose no problem, unless there are several of them. If a word has several `EXIT`s, they have to be treated like a control flow join just before exiting.

Inlining has been the starting point of most Forth native code compilers. In the framework of RAFTS, it can eliminate the call and return overhead, it can improve register allocation and reveal opportunities for further optimization. The important decision in inlining is what to inline [CMCH92, DC94].

If the last useful (i.e., non-stack-manipulation) word is a call, **tail call optimization** can be applied: The call is converted into a jump and the called word returns directly to its callers caller. Of course, this optimization must not be performed if the called word does nasty things to its return address (like throwing it away).

3.2.4 Register allocation

Given the high frequency of calls in Forth, interprocedural register allocation is necessary to make effective use of the register set. Otherwise most of the time would be spent saving and restoring registers at procedure entries and exits, and around calls.

There are a number of good register allocation algorithms, but it is not clear which one is the most appropriate: It should perform good global and interprocedural allocation, but, in order to preserve the interactive feeling, it must not take a long time.

Graph coloring register allocation [Cha82, CH90, Bri92], currently the standard approach, needs a lot of time and space. Hierarchical graph coloring [CK91] looks better and good experiences with it have been reported [Bra95]. Other alternatives are coagulation [Mor91], the Fat Cover algorithm based on hierarchical cyclic interval graphs [HGAM92] and interprocedural allocators [Cho88].

3.2.5 Unknown stack depth

Until now there has been no need for a stack in memory and its stack pointer. All stack items reside in pseudo-registers and the compiler keeps track of the stacks.

Unfortunately, there are definitions that are not as well-behaved as assumed above. They have control flow joins with unequal stack depths, resulting in an unknown stack depth, and, for the whole definition, an unknown stack effect. These definitions are rare, but many applications contain one or two of them. A complete Forth compiler must deal with them.

Some usage patterns that produce an unknown stack depth can be handled automatically without introducing memory stacks and stack pointers, but this is not possible in general. Therefore, we finally have to introduce stacks in memory.

Let us assume that the stack pointer points to the (memory position of the) top-of-stack at the beginning of the definition. Since we do not update the stack pointer, at every point in the definition the stack pointer has a certain, known offset from the *ideal stack pointer* that always points to the (memory position of the) top-of-stack. At a control-flow join which produces an unknown stack depth, the offsets on the joining control-flow edges differ.

The compiler deals with this situation by

- inserting a stack pointer update into one of the joining basic blocks, to make the offsets equal.

- inserting one or more stores of the deepest stack items in registers into that branch, to reduce the depth of the part of the stack in registers to its counterpart; or by inserting instructions that load items from memory, to increase the depth.

Fig. 3.4 shows an example.

In loops, the back edge(s) must be adjusted. In the other cases, the compiler has a free hand. In Forth, a simple, conservative, and (for normal programs) accurate method to handle loops right is to adjust the control-flow edges represented by `dest` control-flow stack items (see Fig. 1.3).

For the remaining cases, a bad selection of the edge to adjust will cause another mismatch on a different control-flow join. So, it is important to select the right edge. An adjustment often entails many moves that cannot be eliminated. If the adjustment is inside a loop, it may be advantageous to store all items before the loop that are not needed in the loop, and to reload them afterwards, to reduce the number of moves in each iteration.

An unknown stack depth inside a definition means an unknown stack effect for the whole definition. A caller of such a definition has to store all

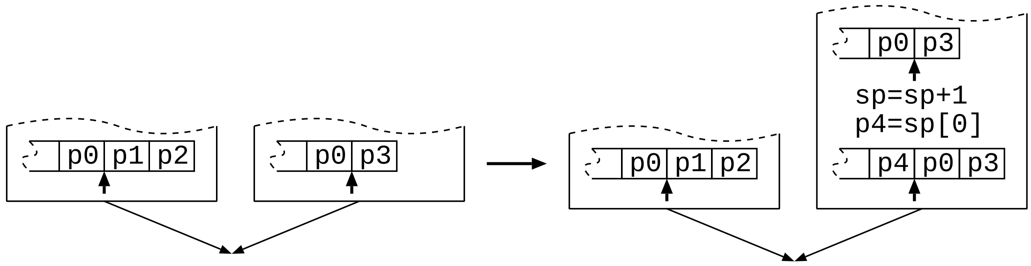

Figure 3.4: Control flow joins with unequal stack depths are handled by adjusting one of the joining basic blocks to match the other. The stack grows downwards in memory.

stack items to the stack that are not passed as parameters, it has to adjust the stack pointer to the offset expected by the callee, and afterwards it has to load the stack items from the stack that it needs, and adjust the stack pointer back. Moreover, it becomes a definition with an unknown stack effect itself. So, a definition with unknown stack effect poisons all its direct and indirect callers.

The only frequent unknown stack effect word in Forth is `?DUP` in contexts like `?DUP 0= IF`, where the `IF` ... `ENDIF` has an unknown stack effect, too. However, the combined stack effect is usually known. The compiler should recognize and handle this situation with ad-hoc logic to avoid the costs of treating this as an unknown stack effect.

3.2.6 Indirect calls

When compiling `EXECUTE` (i.e., an indirect call), the compiler does not know what word will be called and therefore cannot perform interprocedural register allocation. Even worse, the stack effect of the executed word is unknown.

Therefore, when compiling `EXECUTE`, a conventional calling convention is used: A number of items from each stack have to reside in certain registers. The rest is stored in the appropriate places on the stacks; the physical stack pointers have to be updated to reflect their logical values. The storing and loading code and the stack pointer updates can be hoisted to infrequently executed parts using standard techniques [Dha90]. It will have to be determined empirically, how many items of each stack should reside in registers.

When the *execution token*[5] of a word is produced, a special version of the word is generated that conforms to the calling convention. The address of the entry point of this version is the execution token.

[5] An abstract data type that represents the execution semantics of the word.

3.2.7 Dynamic Stack Manipulation (`PICK` and `ROLL`)

Compiling `PICK` and `ROLL` with constant top-of-stack (e.g., `4 PICK`) poses no problem. They can be executed during compilation like other stack manipulation words. But if the top-of-stack value cannot be determined at compile time, this is not possible. There are two solutions:

- The registers are dumped on the stack and the operation is performed in the classical way. In case of `ROLL`, the compiler also has to invalidate the registers and reload the stack items on demand.
- They are translated into something like

  ```
  CASE
    0 OF  0 PICK  ENDOF \ performed at compile time
    1 OF  1 PICK  ENDOF
    ...
    DUP PICK SWAP       \ default: use memory PICK
  ENDCASE
  ```

  with one `OF` for every stack item that resides in a register.

3.2.8 How it fits together

During the processing of the input text, the compiler builds the data flow graphs for the basic blocks and control flow graphs for words. Only the words lower in the call graph have been built yet; therefore, not enough interprocedural information is available and the compiler leaves the rest of the work for a later stage.

The rest of compilation is started by the first call to an uncompiled word. Then this word and all words that it calls are compiled: Unknown stack depths, `EXECUTE`, `PICK` and `ROLL` are resolved, stack pointer updates are inserted, and the code is converted to SSA form. After the optimizations, the code is converted back from SSA form. Then instruction selection, instruction scheduling and register allocation are performed. Finally, the code represented by the data structures of the compiler is translated into machine code and the word is executed.

3.3 Implementation

For his Diplomarbeit [Pir95], Christian Pirker has implemented the basic block part of RAFTS (for the MIPS architecture), with very simple instruc-

	producing	
running in	threaded code	native code
threaded code	1.00	4.98
native code		3.20

Figure 3.5: Relative compile times of different compilers running in different execution techniques

tion selection and instruction scheduling. Register allocation is also performed only at the basic block level, and the stack items reside in memory at the basic block boundaries. The resulting code is not very fast, but that was not the primary goal of that work.

The goal was to make the compiler work, not only for small benchmarks, but good enough to compile itself. This goal has been achieved. The compiler is also very well integrated with the Gforth interpreter on which it is based; native code can transparently call interpreted words and vice versa. The integration is done so well, that it is actually a problem to find out how much interpreted code is executed (which is important in performance measurements).

Christian Pirker is continuing the development of the compiler to produce faster code, supported by the FWF (Austrian science foundation) research project P11231.

3.4 Empirical Results

To get an idea what a complete compiler based on RAFTS can achieve, read Section 4.5 (it should do at least as well as *f2c opt*). Here I just report the results of the simple version described above. The times were taken on a DecStation 2100 (12.5MHz R2000) under Ultrix.

The compiler (running in native code) compiles 3.2 times slower than the compiler of the Gforth system that produces threaded code (compilation speed was not a primary goal). The native-code compiler in native code compiles itself 1.56 times faster than when it is being interpreted (see Fig. 3.5).

We also measured the run-time of the benchmarks used in Section 4.5 (the benchmarks are described in more detail there). The code produced by the RAFTS prototype is a factor of 2 slower than *f2c opt* on most benchmarks. The bad performance of RAFTS on the *bubble* benchmark is due to the fact that the inner loop contains some interpreted words. The speedup over Gforth is 2–3, except for *bubble*. Gforth does a little better with respect to

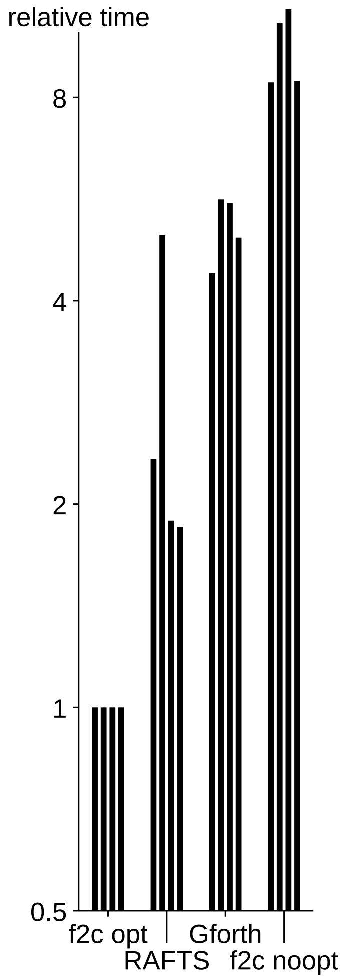

Figure 3.6: The RAFTS prototype in comparison with some other systems on a DecStation 2100

f2c opt than on the 486-66, *f2c noopt* does a little worse.

Chapter 4

Translation to C

Due to C's popularity, there are many sophisticated optimizing compilers for C. Many language implementations (e.g., Eiffel, Modula-3, the first C++ compiler *cfront*, and the SUIF system) have taken advantage of this fact and use C as intermediate language in their compilation process, i.e., they use C as "portable assembler".

In the same way, C can be used as intermediate language for Forth native code compilation (see Fig. 4.1). While this approach requires sacrificing some of Forth's features (mainly interactivity), and is therefore unlikely to become the dominant Forth implementation method, it is useful in the following contexts:

- The customer may require C.
- It can serve as a replacement for the practice of recoding time-critical words in assembly language. The advantages of this approach are that the translation is automatic and the result is machine-independent. However, it requires the ability to call C functions from Forth.
- It can be used for evaluating native code compilers, by comparing their code with the code produced by going through C.
- It can be used for evaluating the usefulness of certain optimization methods (e.g., strength reduction) with respect to existing Forth source

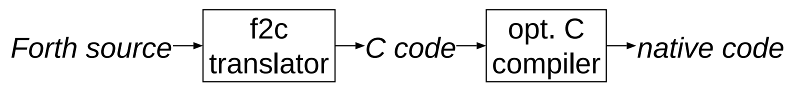

Figure 4.1: Translating Forth to native code via C

```
: max ( n1 n2 -- n )
 2dup < if
   swap drop
 else
   drop
 endif ;
```

Figure 4.2: `max` in Forth

code, by comparing the speed of the code resulting from compiles with and without the optimization.

In the context of the native code compilation project, I am interested primarily in the last two applications.

The emphasis in this chapter is on simplicity. The translation strategy utilizes the optimization capabilities of optimizing C compilers to produce efficient native code. This chapter is a revised version of [EM95].

The contribution of this chapter is a simple translation method that produces efficient native code when combined with an optimizing C compiler.

4.1 Related Work

Several interpretive Forth implementations have been written in C (Cforth, TILE, PFE, ThisForth) and its relatives (Gforth in GNU C [Ert93], UNTIL in C++ [Smi92]). There are also many native code implementations, and several papers about them [Ros86, Alm86, Pay91]. However, to the best of my knowledge, the combination of these ideas, i.e., using C as intermediate language in native code generation for a stack-based language has not been published (before [EM95]), and there has been only one implementation: it comes with Rob Chapman's Timbre/botForth system[1].

Timbre's Forth-to-C translator creates code similar in spirit to the output of a subroutine threaded Forth system that inlines selected primitives, except that it creates C source code, not machine code. It also employs a kind of peephole optimization, but not on the output, but on the Forth input: Like ThisForth, it has a mechanism for recognizing certain Forth sequences, and it has special C code generation rules for these sequences.

Figure 4.2 shows a definition of `max` in Forth. Figure 4.3 shows the C code that the Forth-to-C translator of Timbre V.4 produces for this definition. Unfortunately, current C compilers produce pretty slow assembly code from

[1] Available at http://www.taygeta.com/pub/Forth/Reviewed/timbre.zip

```
void MAX()  /* N1 N2 -- N */
{
  --sp,sp[0]=sp[2];  /* OVER */
  --sp,sp[0]=sp[2];  /* OVER */
  if((Integer)sp[1]<(Integer)sp[0])
    *++sp=-1; else *++sp=0;  /* < */
  if(*sp++)  /* IF */
  {
    sp[1]=sp[0],sp++;  /* SWAP DROP */
  }
  else  /* ELSE */
  {
    sp++;  /* DROP */
  }
}
```

Figure 4.3: `max` translated to C by Timbre

this C code; e.g., Fig. 4.4 contains the code produced with `gcc-2.6.3 -O3 -fomit-frame-pointer` for the Intel architecture. Several features of the C code cause the slow machine code; they all have to do with the inability of C compilers to disambiguate memory references (even if they try, they have little success).

The biggest problem in the code above is that the stack pointer is a global variable. In the absence of analysis of the whole program, the compiler has to assume that the address of this variable has been taken and that every store to memory through a pointer can be a store into the stack pointer, and every read from memory through a pointer could be a read from the stack pointer. So it has to keep the stack pointer in memory up-to-date nearly at all times, and it has to load it from memory after every store through a pointer. A simple way of helping the C compiler would be to use a local stack pointer variable that is initialized from the global stack pointer at the start of the function and after calls and from which the global stack pointer is updated upon leaving the function and before function calls. Then the C compiler could keep the stack pointer in a register at least some of the time.

The next problem is that stack items are accessed through pointers. Again, the C compiler often cannot determine that the stack item stored to memory has not been changed there in the meantime. So, instead of keeping the item in a register, the compiler loads it from memory again. Even if it can determine that the load is unnecessary, it can hardly ever determine that the store of the stack item is unnecessary. So, accessing stack items

```
  addl $-4,_sp
  movl _sp,%edx
  movl 8(%edx),%eax
  movl %eax,(%edx)
  addl $-4,_sp
  movl _sp,%edx
  movl 8(%edx),%eax
  movl %eax,(%edx)
  movl _sp,%edx
  movl 4(%edx),%eax
  cmpl %eax,(%edx)
  jle L7
  addl $4,_sp
  movl _sp,%eax
  movl $-1,(%eax)
  jmp L8
L7:
  addl $4,_sp
  movl _sp,%eax
  movl $0,(%eax)
L8:
  movl _sp,%eax
  addl $4,_sp
  cmpl $0,(%eax)
  je L11
  movl _sp,%edx
  movl (%edx),%eax
  movl %eax,4(%edx)
L11:
  addl $4,_sp
  ret
```

Figure 4.4: Timbre's C code for `max` compiled to Intel assembly

through pointers results in a lot of memory traffic with current C compilers.

Exacerbating these problems are the stack pointer changes in the C code. First of all, with the global stack pointer they result in expensive read-modify-write sequences; second, some C compilers can determine that two accesses through the same pointer access (or do not access) the same item, but they give up if the pointer is changed between the accesses.

4.2 Translation

4.2.1 Efficient C

First of all, there is no such thing as efficient C code per se. How efficient a piece of code is depends on the compiler that is used. E.g., the code that our translation method (described later) produces gives disastrous results with non-optimizing compilers like PCC[2], lcc [FH91b], or the GNU C compiler with optimization turned off. On the other hand, maybe one day (but not in the foreseeable future) there will be compilers that compile the C code produced by Timbre into perfect machine code.

So, what we mean with efficient C code in this paper is C code that is compiled to efficient machine code by a class of compilers commonly known as globally optimizing compilers, e.g., `gcc -O`.

We make extensive use of several features of these compilers: *global register allocation* [Cha82] tries to put as many local variables into registers as possible[3]. *Live range analysis* [ASU86] allows the compiler to put several variables into the same register, if they do not contain subsequently used (*live*) data at the same time (i.e., their live ranges do not overlap); conversely, it allows putting different live ranges of one variable in different registers. *Copy propagation* [ASU86] optimizes away simple assignments between variables (copies) by assigning the variables to the same register. These optimizations allow us to introduce unnecessary copies and additional variables for free (but the number of simultaneously live variables is limited), but it is not necessary to use additional variables for register allocation purposes. Another optimization, dead code elimination [ASU86], removes all code whose results are not used; this optimization is useful for Forth code containing `drop`, in particular frequent sequences like `r> drop`.

Current C compilers are not good at keeping global variables or values accessed through pointers in registers.

4.2.2 Stack representation

Given the limitations of the available C compilers, we have to represent stack items with C's local variables, so they will end up in registers. The correspondence between stack items and local variables is shown in Fig. 4.5:

[2]PCC was the C compiler distributed with most Unix systems 15 years ago.

[3]In optimization terminology *local* means "within a basic block", *global* means "within a procedure", and everything beyond is called *interprocedural.* By contrast, in programming language terminology *local* means "within the procedure" and *global* means "in the whole program".

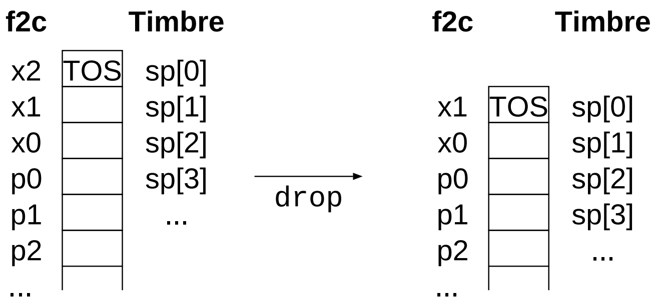

Figure 4.5: Correspondence between the Forth stack and C entities in our translator (f2c) and in Timbre.

a local variable represents a stack item with a certain offset from the stack bottom during the whole C function.[4] E.g., the stack item on the top-of-stack upon function entry is called `p0`, the item below it `p1` etc.; the stack item above `p0` is called `x0`, the next one `x1` etc. This scheme ensures that stack items that are not affected by an operation do not have to be copied around between local variables (although these copies would not hurt, given a good copy propagator).

4.2.3 Primitives

Let us assume that the top of stack resides in `x3`. Then the translation of `+` might look like this:

```
{
  Cell n1=x2;
  Cell n2=x3;
  Cell n;
  n = n1+n2;
  x2 = n;
}
/* top of stack now: x2 */
```

This sequence looks quite bulky, but a good optimizing C compiler compiles it to one instruction, and sometimes it can even compile the C sequence for several primitives into one instruction (e.g., `20 + @` on most processors).

[4] In [EM95], I advocated not reusing locals, i.e., introducing a new local every time a stack item is written. The reason I gave was that the reuse might restrict register allocation and copy propagation on some compilers. However, a translator that does not reuse locals is significantly more complex, and the C compilers we have used (`gcc` and the MIPS/Ultrix `cc`) work well on code that does reuse locals.

I propose this translator output because it is simple to generate from specifications of primitives. In fact, we can reuse the specifications used for generating the Gforth interpreter [Ert93]. E.g., the specification for `+` is

```
+    n1 n2 -- n    core    plus
n = n1+n2;
```

The first line contains useful information, most notably, the name of the primitive and its stack effect (including the names and types[5] of the stack items), while the other lines (in the present case the second line) contain the C code for the primitive.

The translation works like this: First, the names in the stack effect are declared as C variables; the input variables are initialized with the top stack items. Next comes a verbatim copy of the C code for the primitive. The translator adjusts its (translation-time) stack depth counter. Finally, the result variables are copied to stack item variables.

4.2.4 Sequences

A sequence of primitives can be translated simply into a sequence of translations. E.g., if we want to compile another `+`, we simply append the following code to the code above:

```
{
  Cell n1=x1;
  Cell n2=x2;
  Cell n;
  n = n1+n2;
  x1 = n;
}
/* top of stack now: x1 */
```

4.2.5 Definitions

A Forth colon definition is translated into a C function. In this section we consider only colon definitions with fixed stack effects.

The translator determines the stack effect of the whole definition by keeping track of the deepest stack item accessed, in combination with the stack depth count: If the deepest accessed item is pn, and the top of stack item at the `EXIT`(s) is z, then the stack effect is pn...`p0` -- pn...z. If there are several

[5]For specifying the types, I use the good old FORTRAN convention of deriving it from the first letter(s) of the name. E.g., names starting with `n` specify signed integer cells (corresponding to the C type `Cell`).

result values, they are bundled into a C struct (small structs are returned in registers by some C compilers, e.g., `gcc`). E.g., the C function for the word `foo ( n1 n2 n3 -- n4 n5 )` is:

```
Two_cells foo(Cell p0, Cell p1, Cell p2)
{
  Two_cells result;
  Cell x0, x1, ....;

  /* top of stack now: p0 */
  ... /* C code for the definition */
  /* top of stack now: p1 */
  result.cell1=p1;
  result.cell2=p2;
  return result;
}
```

A call of foo would look like this:

```
  /* top of stack now: x3 */
  {
    Two_cells d=foo(x3,x2,x1);
    x2=d.cell1;
    x1=d.cell2;
  }
  /* top of stack now: x2 */
```

The parameter order is more or less arbitrary. Having the top-of-stack first ensures that in cases like

```
: foo
   bar
   ... ;
```

the parameters need not be moved to new registers before calling `bar` on machines that pass parameters through registers. However, one can also construct cases, where having the top-of-stack last is better. It has to be determined empirically which convention is better for real-world Forth code.

Turning definitions into C functions is simple, but makes tricks like `r> drop exit` almost impossible to translate (but they do not conform to ANS Forth anyway). It also has a performance impact, because most C compilers are not as optimized for reducing call overhead as much as you would like for a Forth compiler. A C compiler with automatic inlining and interprocedural register allocation should perform well, however. Alternatively, we could build a certain amount of inlining into the Forth-to-C translator.

4.2.6 Control Structures

ANS Forth allows the creation of arbitrary control structures (see Fig. 1.3). Because it is hard and, in the context of sophisticated C compilers, unrewarding[6], to translate arbitrary control structures into structured C, we translate the control structure words into `goto`s and labels:

Forth	C
`THEN`, `BEGIN`	*label*:
`AHEAD`, `AGAIN`	`goto` *label*;
`IF`, `UNTIL`	`if (`x`==0) goto` *label*;

Control-flow stack items contain a label number and a stack depth count. At control-flow joins (`THEN`, `BEGIN`) the stack depths of both joining control flow edges have to be equal (otherwise the stack depth at the join is unknown, see Section 4.2.11). The consumers of control-flow stack items check this. Because the respective stack items of both joining control-flow paths are already in the same variables, there is no need to reconcile the stack state at control flow joins (in contrast to direct native code generation, see Section 3.2.2)[7].

4.2.7 Return Stack

The return stack is handled like (a restricted version of) the data stack: Its items are kept in C variables, and the translator keeps track of the stack depth. In contrast to the data stack, a definition may only access return stack items that it pushed on the return stack itself (on the data stack, such items are represented by the `x`n variables).

E.g., a translation of `>r` looks like this:

```
/* stack: ... x0, return-stack: ... r0 */
{
  Cell n=x0;
  r1=n;
}
/* stack: ... p0, return-stack: ... r1 */
```

The floating-point stack is managed like the data stack.

[6]Optimizing C compilers usually use the same intermediate representation for control structures built with structured programming constructs and for equivalent control structures built with `goto`s. While there are simpler optimization methods for structured code, they apparently only pay off if the compiler does not have to deal with `goto`s at all.

[7]This simple handling of control flow is the main advantage of the stack representation described in Section 4.2.2 over one that does not reuse locals.

4.2.8 Names

Forth has a different, more complex and more powerful name space structure than C. Moreover, Forth allows names that are not legal C identifiers. Therefore, we cannot use the Forth names directly in the C code. A simple way out is to derive the C name from a pointer to a data structure representing the word in the translator (e.g., by printing it in base 36).

However, in practice we prefer a name that is as close to the original as possible. This can be achieved by converting special characters in names into letter sequences and by appending some digits if that is necessary to avoid conflicts.

4.2.9 Locals

Forth local variables can be translated simply into C locals. If sophisticated scoping behaviour as in, e.g., [Ert94a], is desired, it is probably easiest to define the locals on the C level for the whole function and to avoid name clashes by renaming.

4.2.10 Other Word Types

Variables and `CREATE`d words are translated into global (or `static`) C variables of appropriate size. E.g., `5 variable flip` translates into:

```
Cell flip[]={5};
```

When used in a definition, they are translated like primitives with similar stack effect, e.g., an occurence of `flip` is translated into

```
{
  Cell n;
  n= (Cell)flip;
  x2=n;
}
/* top of stack now: x2 */
```

`DOES>`-parts are translated into C functions like colon definitions—their top-of-stack parameter is the address of the word. Accordingly, using a word defined with a `CREATE..DOES>`-word translated into a call of the C function for the `DOES>`-part, with the address of the C variable produced by the `CREATE` as top-of-stack parameter.

4.2.11 Unknown stack depth

The methods presented until now cannot handle unknown stack depths. An unknown stack depth shows up at control-flow joins as unequal stack depth of the joining control-flow edges.

Given that the translation to C cannot cover the whole Forth language in a satisfying way, it probably does not pay off to translate definitions with unknown stack depths. If, however, such a translation is desired, the methods discussed in Section 3.2.5 can be applied.

4.2.12 Recursive Definitions, indirect calls, etc.

Computing the stack effect of a recursive word is a little different from other words: With the normal method we would need the stack effect of the word for computing it. However, if we assume that the recursive word has a known stack effect, the stack effect is the same as the stack effect of the non-recursing path(s) through the word. Using this stack effect, the translator can check the validity of the assumption in another pass through the definition. If the definition turns out to have an unknown stack effect, it can be translated like any other such definition.

`EXECUTE` and deferred words pose a similar, but harder problem: While all words executed by a specific `EXECUTE` usually have the same stack effect, the translator does not know that stack effect. One solution for this problem is to treat `EXECUTE` and deferred words like words with unknown stack effect, the other solution is to use annotations provided by the programmer to specify a stack effect.

Execution tokens are represented by C function pointers, and `EXECUTE` just performs a call to the pointed-to function. This representation of execution tokens implies that the translator has to create a C function for each `CREATE`d word, variable, or constant for which an execution token is produced.

4.2.13 Cross-compilation problems

One of the more remarkable features of Forth is the removal of the strict division between compile-time and run-time. While typical Forth-to-C translators will have similar restrictions in this respect as some Forth cross-compilers, with some effort these restrictions can be circumvented: Modern operating systems offer dynamic linking of object code. A Forth system based on a Forth-to-C translator could produce C code, then, at the end of the definition, it would invoke the C compiler, and dynamically link the resulting object module. This approach would require an unhealthy amount of patience from the users, but otherwise it would be a full Forth system.

Many implementors of Forth-to-C translators will not want to go to such lengths and will implement the translator in the context of an existing Forth system. In such a setting, compilation and execution can be mixed during the translation stage (i.e., while running on the normal Forth system), while only execution is possible during the execution of the translated and compiled program.

In such cross-translation systems, there is also the problem that addresses cannot be compiled as literals or stored into variables and data structures at compile time, because the addresses are different at run-time. One solution to this problem is to require using typed words (`ALITERAL`, `A!`, `A,` etc.) for these operations, and using the type information for the relocation. A more convenient, but less portable (i.e., OS-dependent) solution is to load the whole image of the translating Forth system to the same address as during the translation. In such a system, variables etc. are not represented as C variables, they behave just like literals.

4.3 Example

A Forth definition of `max ( n1 n2 -- n )` (Fig. 4.2) is translated into C (Fig. 4.6), then `gcc-2.6.3 -O3 -fomit-frame-pointer` compiles the C code into Intel assembly (Fig. 4.7). Note that three of the six instructions here are due to calling overhead and can be eliminated with inlining.

4.4 Implementation

Martin Maierhofer has implemented a proof-of-concept Forth-to-C translator in about one man-month, with no prior knowledge of Forth. The translator is based on the Gforth system. The source can be found at http://www.complang.tuwien.ac.at/forth/forth2c.tar.gz (if you only have ftp: ftp.complang.tuwien.ac.at/pub/forth/forth2c.tar.gz).

Basically, the translator hooks itself into some words central to compilation: `COMPILE,`, `LITERAL`, the basic control structure words, `:`, `;`, `CREATE` and friends. It also hooks into words that compile in-line data, like `S"`.

A program is translated by loading it into Gforth. If the translator is turned on, Gforth will not only compile the program into threaded code, but, as a side effect, it will also produce a file containing the C code. In this

```
Cell max(Cell p0, Cell p1)
{
Cell _c_result;
Cell x0;
Cell x1;

   /* stack now: ... p1 p0 */
   {   /* 2dup */
      Cell n1, n2;
      n1 = p0;
      n2 = p1;
      p1 = n2;
      p0 = n1;
      x0 = n2;
      x1 = n1;
   }
   /* stack now: ... p1 p0 x0 x1 */
   {   /* less-than */
      Cell n1, n2, n;
      n1 = x1;
      n2 = x0;
      n = FLAG(n2 < n1); /* #define FLAG - */
      x0 = n;
   }
   /* stack now: ... p1 p0 x0 */
   if (!x0) goto label0;
   /* stack now: ... p1 p0 */
   {   /* swap */
      Cell n1, n2;
      n1 = p0;
      n2 = p1;
      p1 = n1;
      p0 = n2;
   }
   /* stack now: ... p1 p0 */
   {   /* drop */
   }
   /* stack now: ... p1 */
   goto label1;
   label0:
   /* stack now: ... p1 p0 */
   {   /* drop */
   }
   /* stack now: ... p1 */
   label1:
   /* stack now: ... p1 */
   {   /* exit */
      _c_result = p1;
      return (_c_result);
   }
}
```

Figure 4.6: `max` translated to C

```
  movl 4(%esp),%edx
  movl 8(%esp),%eax
  cmpl %edx,%eax
  jge L5
  movl %edx,%eax
L5:
  ret
```

Figure 4.7: `max` in Intel assembly

way, the full power of Gforth can be used during the translation process.[8]

Since the Forth compiler makes only one pass through a definition, everything must be done in that pass. There is just one problem: The translator must make a pass through the whole definition to compute the stack effect, and it needs to know the stack effect to create the C function header, so it can only start outputting C code at the end of the definition. The solution is to generate the text of the body of the function into a buffer in memory, and write it to the output file as soon as the stack effect is known (at the end of the definition).

The translator must remember the stack effect of a definition somewhere. The best place would be in the header of the definition. Unfortunately, this would require some surgery in the internals of Gforth. Therefore, the translator keeps these informations in a separate wordlist under the same name as the definition. This means, of course, that every definition name must be used only once. But then this is also enforced by the mapping from Forth names to C names employed by the prototype translator.

In addition to the normal control flow information, each control-flow stack item stores the label number and the stack depths at the point where the control flow item was generated.[9] Our translator stores this extended control flow information on a separate, user-defined stack (Gforth stores its information on the data stack).

While it would have been nice to generate the translation of primitives automatically from Gforth's primitive specifications, as suggested in Section 4.2.3, the translator does not employ this approach, but uses handwritten primitives translation routines.

The most important limitations of this prototype translator are that it cannot translate words defined with `DOES>`, initialized `CREATE`d words, `EXECUTE`, many recursive definitions, definitions with variable stack effects or

[8]Well, at least in theory; in practice, the translator has a few restrictions described in the file BUGS.F2C.

[9]Our translator is somewhat buggy in this area, it does not handle `THEN` correctly.

definitions that use the floating-point stack or locals. Some of these restrictions would be easy to remove, some of them would take more effort.

4.5 Empirical Results

Due to the restrictions of our translator we could not use it on realistically sized benchmarks, only on small ones. On the other hand, this restriction to small benchmarks means that it is easier to compare with other Forth systems.

The benchmarks we used were the ubiquitous Sieve (counting the primes < 16384 a thousand times); bubble-sorting (6000 integers) and matrix multiplication (200×200 matrices) come from the Stanford integer benchmarks (originally in Pascal, but available in C[10]) and have been translated into Forth by Martin Fraeman and included in the TILE Forth package. These three benchmarks share one disadvantage: They have an unusually low amount of calls. To benchmark calling performance, we computed the 34^{th} Fibonacci number using a recursive algorithm with exponential run-time complexity.

The measured systems fall into several classes:

C-based native code

f2c opt the output of our prototype translator compiled with `gcc-2.6.3 -O3 -fomit-frame-pointer`.

Timbre the output of the Forth-to-C translator included in the Timbre V.4 distribution, also compiled with `gcc-2.6.3 -O3 -fomit-frame-pointer`.

f2c noopt the output of our prototype translator compiled with `gcc-2.6.3` without optimization.

C hand-coded C programs (the original C versions in case of the Stanford benchmarks) compiled with `gcc-2.6.3 -O3 -fomit-frame-pointer`.

Native code compilers, typically employing the traditional technique of combining subroutine threading with inlining and peephole optimization.

bigForth bigForth 386 (v1.20β) by Bernd Paysan [Pay91].

iForth iForth 1.06 by Marcel Hendrix.

[10] The C version and Martin Fraeman's original translations to Forth can be found at ftp.complang.tuwien.ac.at/pub/forth/stanford-benchmarks.tar.gz

NT NCC LMI's NT Forth NCC (written by Tom Almy).

Interpreters written in assembly language

Win32F Win32Forth 1.2093 by Andrew McKewan and Tom Zimmer.

NT Forth LMI's NT Forth (beta, May 1994).

eforth Marcel Hendrix' port of eforth to Linux.

eforth opt Marcel Hendrix added peephole optimization of intermediate code [Bad95, Sch92] to his port of eforth.

Interpreters written in portable languages

Gforth Gforth 0.1beta, written in GNU C [Ert93] (compiled with `gcc-2.6.3`, default flags and `-DFORCE_REG -DDIRECT_THREADED`).

PFE pfe-0.9.14 by Dirk Zoller, written in ANSI C (compiled with `gcc-2.6.3` with the default configuration)

thisForth ThisForth Beta written in C by Wil Baden (compiled with `gcc-2.6.3 -O3 -fomit-frame-pointer`); it employs peephole optimization of intermediate code.

TILE TILE Release 2.1, written in C by Mikael Patel (compiled with `gcc-2.6.3 -O3 -fomit-frame-pointer`).

Win32Forth, NT Forth and its NCC were benchmarked under Windows NT, bigForth and iForth under DOS/GO32, all other systems under Linux. The results for Win32Forth, NT Forth and its NCC were provided by Kenneth O'Heskin, those for iForth and eForth (with and without peephole optimization of intermediate code) by Marcel Hendrix. All measurements were performed on PCs with an Intel 486DX2/66 CPU with 256K secondary cache with similar memory performance. The times given are the median of three measurements of the user time (the system time is negligible anyway).

Figure 4.8 shows the time that the systems need for the benchmarks, relative to the time of *f2c opt* (or, in other words, the speedup factor that our translator (and GCC) achieves over the other systems). Empty entries indicate that I did not succeed in running the benchmark on the system.

The result of Timbre's Forth-to-C translator is slow, as expected (since Timbre does not have `DO..LOOP` and friends, I could only measure fib, but I think this result is representative). Combining our translator with a non-optimizing GCC results in code that is even slower than the interpretive Gforth system, confirming the points I make about efficient C code in Section 4.2.1. Hand-coded C is between 14% faster and 10% slower than the

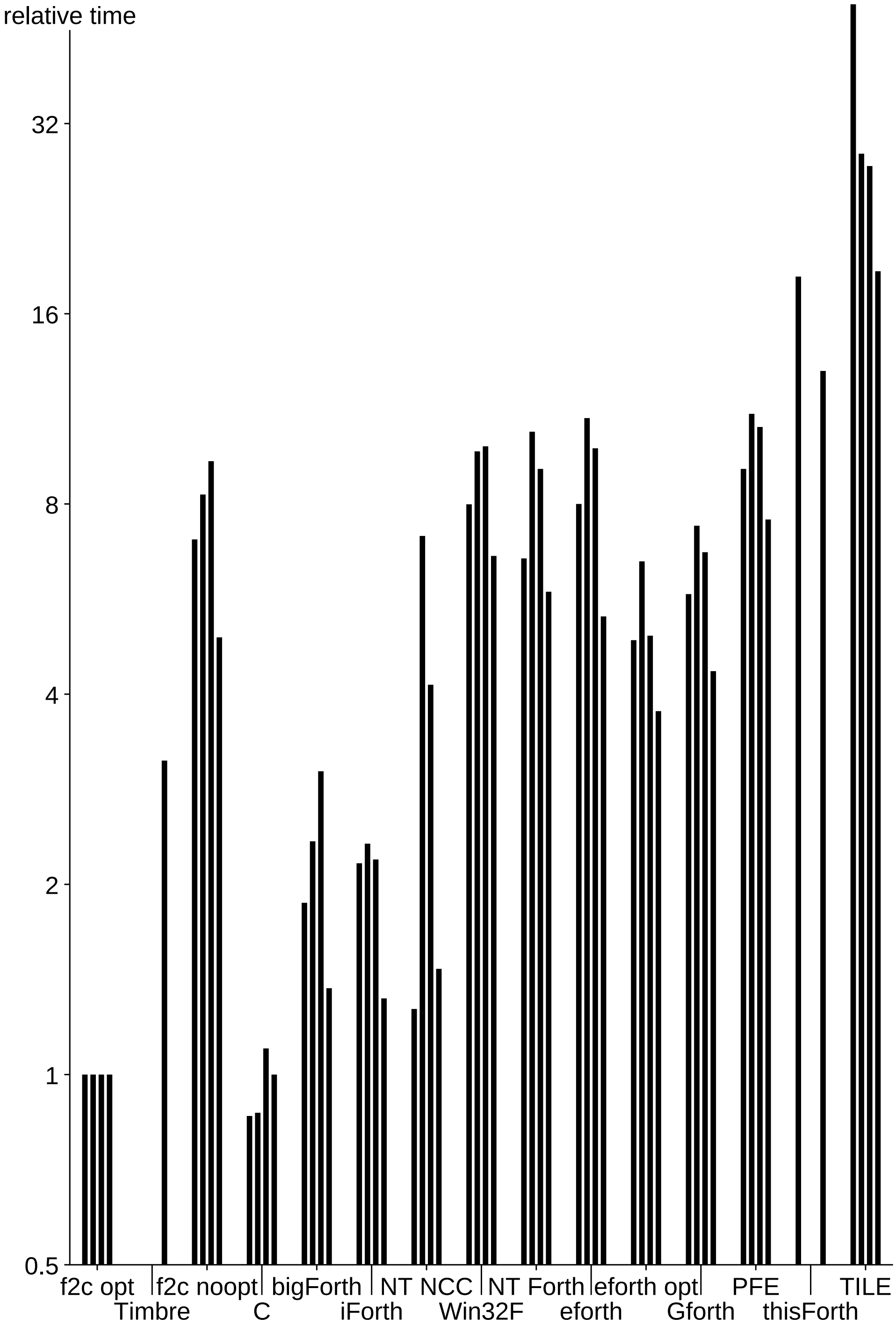

Figure 4.8: Time needed by various systems for several benchmarks, relative to the output of our prototype Forth-to-C translator (f2c opt.); lower means faster.

output of the Forth translator. I was a little surprised by the matrix multiplication result, where the code translated to Forth and back was faster than the original. Closer inspection showed that, in translating from C to Forth, an optimization had been performed, which the C compiler does not perform (it is an interprocedural optimization that requires interprocedural alias analysis) and which reduced the amount of memory accesses; this is probably responsible for the speedup, maybe combined with vagaries such as instruction cache alignment.

BigForth and iForth achieve a speedup of about 3 over the fastest interpreters, but there is still a lot of room for improvement (at least a factor of 1.3–3). Even on the fib Benchmark, which should be the strong point of Forth compilers, *f2c opt* was better. The results of NT Forth NCC are a bit worse on average and have a big variance (a speedup of 1–5 over Gforth, 1.2–7.2 times slower than f2c opt.). These results show that researching better native code generation techniques is not just a waste of time and that there is still a lot to be gained in this area. These results also show that the following statement has not become outdated yet: “The resulting machine code is a factor of two or three slower than the equivalent code written in machine language, due mainly to the use of the stack rather than registers.” [Ros86]

The interpretive systems written in assembly language (except *eforth opt*) are, surprisingly, slower than Gforth, although the code produced by `gcc-2.6.3` for Gforth is less than optimal, in particular with respect to register allocation. One important reason for the disappointing performance of these systems is probably that they are not written optimally for the 486 (e.g., they use the `lods` instruction). *eforth opt* demonstrates that peephole optimization of intermediate code offers substantial gains. *eforth opt* is 3.5–6.5 times slower than *f2c opt*. We can expect even better results if the baseline interpreter is more efficient (e.g., Gforth).

Gforth is 4–7.5 times slower than *f2c opt*, *PFE* 7.5–11 times. The slowdown of PFE with respect to Gforth can be explained with the self-imposed restriction to standard C (although the measured configuration of PFE uses a GNU C extension: global register variables), which makes efficient threading impossible (PFE uses indirect call threading, see Fig. 2.3). *ThisForth* and *TILE* were obviously written with a certain negligence towards efficiency issues and the limited optimization abilities of state-of-the-art C compilers, resulting in a slowdown factor of more than 49 for TILE on the Sieve.

I not only measured run-time, but also code size and compile time (see Fig. 4.9). For threaded code (interp. size), I measured the space alloted in the Gforth system during compilation of the program and subtracted the alloted data; i.e., the interpreted code size includes the space needed for

	interp. size	.o size	size ratio	compile time	source lines	C lines
sieve	418	272	1.54	1.10s	25	482
bubble	1020	748	1.36	1.60s	72	1100
matmul	784	412	1.90	1.40s	55	793
fib	140	140	1.00	0.90s	10	169

Figure 4.9: Code size and compile time

headers. Gforth uses one cell (32 bits in the measured system) per compiled word, two cells for the code field and it pads the header such that the body is maximally (i.e., 8-byte-) aligned. For the size of the machine code produced by the translator/compiler combination (.o size), I used the sum of the text and data sizes produced by the Unix `size` command, as applied to the object (`.o`) file. This does not include the size of the symbol table information included in the object file (which is easy to strip away after linking). The data size in the object file does not include the alloted space, as that is allocated later at run-time.

The code size measurements dispell another popular myth, that of the inherent size advantage of stack architecture code and of the bloat produced by optimizing C compilers. While a comparison of a header-stripping 16-bit Forth with a RISC (≈50% bigger code than CISCs) would give a somewhat different result, the reported size differences of more than an order of magnitude need a different explanation: differences in the functionality of the software and different software engineering practices come to mind.

For the compile time measurements, Fig. 4.9 only displays the user time needed by GCC to compile and link the program. The system time was constant at 0.6s. The compilation to Gforth's interpreted code needed a negligible amount of time; the translation to C also vanished in the measurement noise, although it was not written for speed and although the present implementation should be much slower than normal Forth compilation. The compile time data indicate that, after a startup time of about 1.4s (user+system), GCC compiles about 90 lines of Forth code (1500 lines of translator output) per second. Interestingly, less than one byte of machine code is generated per line of C code.

Chapter 5

Conclusion

Stack-based languages can be implemented efficiently on register architectures by caching top-of-stack items in registers and by combining stack pointer updates. Although register sets are small compared to data caches, they are big enough to capture almost all stack accesses. Such a stack cache has to be managed by software.

For virtual machine interpreters, the cache can be managed by viewing it as finite state machine. There is a large variety of stack cache organizations. Stack caching can be employed in two ways: In dynamic stack caching the interpreter keeps track of the state of the cache. A copy of the complete interpreter has to be kept for every state of the cache, making only cache organization with few states feasible. Moreover, on many processors dynamic stack caching increases instruction dispatch time, eliminating the speed advantage of stack caching. In static caching, the compiler keeps track of the cache state. This allows using organizations with more states, it allows fast direct threading, and stack manipulation operations can often be optimized away completely. But there is a bit of overhead for making the state conform to calling conventions and reconciling the cache states on control flow joins.

My native code compilation technique transforms the stack-based code into data flow graphs and static single assignment form. Then it can apply standard code generation and optimization techniques. In particular, it puts stack items in pseudo-registers, and global (or interprocedural) register allocation puts them into machine registers. After the transformation, most of the code does not even need the usual memory stacks and their stack pointers, and therefore it does not need stack-pointer updates. For the remaining code, there is at most one stack pointer update and a few memory accesses per control flow join.

Translation to C is similar to native code compilation. However, the translator takes advantage of the optimizer of the C compiler to make the job simple. Optimizing C compilers translate C's local variables into registers. So, the translator represents stack items with local variables. Definitions (procedures) are translated into C functions, and parameters passed over the stack are translated into C function arguments. This translation method is so simple that someone new to Forth has implemented it (in a Forth system) in one person-month.

Caching the data stack in Forth interpreters can eliminate about two real machine instructions per (original) virtual machine instruction. I expect similar savings for other stack-based languages. With the assumption that half of the return stack overhead can be eliminated, I expect 20%–31% speedup on a DecStation 3100 using static stack caching. Dynamic stack caching does not pay off on this machine, due to the increased instruction dispatch cost.

A native code compiler based on the techniques described here is in development. Based on experiments with the prototype Forth-to-C translator I expect the final native-code compiler to produce code that is at least 1.3–3 times faster on a 486-66 than native code produced by Forth compilers that employ traditional techniques (subroutine threading with inlining and peephole optimization).

Our prototype Forth-to-C translator, combined with GCC (with optimization) produces code that is more than three times faster (on a 486-66) than a translator that produces straight-forward C code, combined with the same compiler. The effectiveness of the translation method becomes particularly obvious in comparison with handwritten C code (compiled with the same C compiler), which is between 15% faster and 9%slower the translated code. Moreover, the resulting code is smaller than the intermediate code of a 32-bit Forth interpreter.

Acknowledgements

Christian Pirker has implemented the basic-block-level RAFTS prototype that was used for the results in Section 3 [Pir95]; Since February 1996 the FWF (Austrian Science Foundation) is supporting this work as project P11231, "Compilation of Stack-based Languages". Martin Maierhofer has implemented the prototype Forth-to-C translator that provided the results reported in Section 4. These implementations also led to a better understanding of some of the issues involved (e.g., whether the data stack should be used for building the data flow graph; or, whether a new C variable should be introduced for every item pushed on the stack).

Bernd Paysan and Jens Wilke wrote substantial parts of the Gforth interpreter that was used for all parts of this work.

Marty Fraeman, John Hayes and Chris Bailey discussed program behaviour and their experiences with the random walk model with me.

Tom Almy, Mike Haas, Mike Hore and Bernd Paysan discussed their native code Forth compilers with me.

Martin Fraeman provided the C versions of the Stanford benchmarks. Kenneth O'Heskin kindly produced the benchmark results for Win32Forth, NT Forth, and NT Forth NCC. Marcel Hendrix provided the iForth and eForth results.

Felix Beer, Robert Bernecky, Manfred Brockhaus, Marcel Hendrix, Andreas Krall, Ulrich Neumerkel, Herbert Pohlai, Franz Puntigam, Konrad Schwarz, and the EuroForth and SIGPLAN PLDI'95 referees provided valuable comments on the papers that were the basis of this thesis.

Manfred Brockhaus and Andreas Krall also commented on draft versions of this thesis.

Bibliography

[Alm86] Thomas Almy. Compiling Forth for performance. *Journal of Forth Application and Research*, 4(3):379–388, 1986.

[ANS94] American National Standards Institute. *American National Standard for Information Systems: Programming Languages: Forth*, 1994. Document X3.215-1994.

[AS92] Kristy Andrews and Duane Sand. Migrating a CISC computer family onto RISC via object code translation. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 213–222, 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bad90] Wil Baden. Virtual rheology. In *FORML'90 Proceedings*, 1990.

[Bad95] Wil Baden. Pinhole optimization. *Forth Dimensions*, 17(2):29–35, 1995.

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[Bla77] Russell P. Blake. Exploring a stack architecture. *IEEE Computer*, 10(5):30–39, May 1977.

[Bra95] Marc Brandis. Register allocation using graph coloring. In `c.c` [`c.c`].

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, Houston, 1992.

[`c.c`] `comp.compilers`. Usenet Newsgroup; archives available from ftp://primost.cs.wisc.edu.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, 1982.

[Cho88] Fred C. Chow. Minimizing register usage penalty at procedure calls. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94, 1988.

[CK91] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 192–203, Toronto, 1991.

[CMCH92] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience*, 22(5):349–369, May 1992.

[CU89] Craig Chambers and David Ungar. Custumization: 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.

[DC94] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *Conference on Lisp and Functional Programming*, pages 273–282, 1994.

[Dew75] Robert B.K. Dewar. Indirect threaded code. *Communications of the ACM*, 18(6):330–331, June 1975.

[Dha90] Dhananjay Madhav Dhamdhere. A usually linear algorithm for register assignment using edge placement of load and store instructions. *Computer Languages*, 15(2):83–94, 1990.

[DM82] David R. Ditzel and H. R. McLellan. Register allocation for free: The C machine stack cache. In *Symposium on Architectural Support for Programming Languages and Systems*, pages 48–56, 1982.

[DV90] Eddy H. Debaere and Jan M. Van Campenhout. *Interpretation and Instruction Path Coprocessing*. The MIT Press, 1990.

[EM95] M. Anton Ertl and Martin Maierhofer. Translating Forth to efficient C. In *EuroForth '95 Conference Proceedings*, Schloß Dagstuhl, Germany, 1995.

[Ert92] M. Anton Ertl. A new approach to Forth native code generation. In *EuroForth '92*, pages 73–78, Southampton, England, 1992. MicroProcessor Engineering.

[Ert93] M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Láznè (Marienbad), 1993.

[Ert94a] M. Anton Ertl. Automatic scoping of local variables. In *EuroForth '94 Conference Proceedings*, pages 31–37, Winchester, UK, 1994.

[Ert94b] M. Anton Ertl. Stack caching for interpreters. In *EuroForth '94 Conference Proceedings*, pages 3–12, Winchester, UK, 1994.

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

[FH91a] Christopher W. Fraser and David R. Hanson. A code generation interface for ANSI C. *Software—Practice and Experience*, 21(9):963–988, September 1991.

[FH91b] Christopher W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, October 1991.

[FH95] Christopher Fraser and David Hanson. *A Retargetable C compiler: Design and Implementation*. Benjamin/Cummings Publishing, 1995.

[FHP91] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — *Fast Optimal Instruction Selection and*

Tree Parsing, 1991. Available via anonymous `ftp` from `kaese.cs.wisc.edu`, file `pub/burg.shar.Z`.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Hay89] John Hayes. Design tradeoffs in a top of stack cache. Unpublished, 1989.

[HFWZ87] John R. Hayes, Martin E. Fraeman, Robert L. Williams, and Thomas Zaremba. An architecture for the direct execution of the Forth programming language. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pages 42–48, 1987.

[HGAM92] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In *Compiler Construction (CC'92)*, pages 176–191, Paderborn, 1992. Springer LNCS 641.

[HL89] John Hayes and Susan Lee. The architecture of the SC32 Forth engine. *Journal of Forth Application and Research*, 5(4):493–506, 1989.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach.* Morgan Kaufman Publishers, 1990.

[HS85] Makoto Hasekawa and Yoshiharu Shigei. High-speed top-of-stack scheme for VLSI processor: A management algorithm and its analysis. In *International Symposium on Computer Archictecture (ISCA)*, pages 48–54, 1985.

[KB92] Andreas Krall and Thomas Berger. Fast Prolog with a VAM_{1p} based Prolog compiler. In *Programming Language Implementation and Logic Programming (PLILP '92)*, pages 245–259. Springer LNCS 631, 1992.

[KH92] G. Kane and J. Heinrich. *MIPS RISC Architecture.* Prentice-Hall, 1992.

[KKR65] H. Kanner, P. Kosinski, and C. L. Robinson. The structure of yet another ALGOL compiler. *Communications of the ACM*, 8(7):427–438, July 1965.

[Kli81] Paul Klint. Interpretation techniques. *Software—Practice and Experience*, 11:963–973, 1981.

[Kna93] Peter J. Knaggs. *Practical and Theoretical Aspects of Forth Software Development*. PhD thesis, School of Computing and Mathematics, University of Teesside, Middlesbrough, Cleveland, UK, March 1993.

[Kog82] Peter M. Kogge. An architectural trail to threaded-code systems. *IEEE Computer*, pages 22–32, March 1982.

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

[Koo92] Philip J. Koopman, Jr. A preliminary exploration of optimized stack code generation. In *1992 Rochester Forth Conference*, 1992.

[KPR91] C. W. Keßler, W. J. Paul, and T. Rauber. A randomized heuristic approach to register allocation. In *Programming Language Implementation and Logic Programming (PLILP)*, pages 195–206, Passau, 1991. Springer LNCS 528.

[Kra83] Glen Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983.

[LDSM80] David Landskov, Scott Davidson, Bruce Shriver, and Pattrick W. Mallet. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, September 1980.

[Mor91] W. G. Morris. CCG: A prototype coagulating code generator. In *SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 45–58, 1991.

[Pay91] Bernd Paysan. Ein optimierender Forth-Compiler. *Vierte Dimension*, 7(3):22–25, September 1991.

[Pir95] Christian Pirker. Übersetzung von Forth in Maschinensprache. Master's thesis, Technische Universität Wien, 1995.

[Pit87] Thomas Pittman. Two-level hybrid interpreter/native code execution for combined space-time efficiency. In *Symposium on Interpreters and Interpretive Techniques (SIGPLAN '87)*, pages 150–152, 1987.

[PLG88] Eduardo Pelegrí-Llopart and Susan L. Graham. Optimal code generation for expression trees: An application of the BURS theory. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 294–308, 1988.

[Pöi94] Jaanus Pöial. Forth and formal language theory. In *EuroForth '94 Conference Proceedings*, pages 47–52, Winchester, UK, 1994.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.

[RCM93] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. The evolution of Forth. In *History of Programming Languages (HOPL-II) Preprints*, pages 177–199, 1993. SIGPLAN Notices 28(3).

[RH92] Norman Ramsey and David R. Hanson. A retargetable debugger. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 22–31, 1992.

[Ros86] Anthony Rose. Design of a fast 68000-based subroutine-threaded Forth with inline code & an optimizer. *Journal of Forth Application and Research*, 4(2):285–288, 1986. 1986 Rochester Forth Conference.

[Sch92] Udo Schütz. Optimierung von Fadencode. In *FORTH-Tagung*, Rostock, 1992. Forth Gesellschaft e.V.

[SK93] Bill Stoddart and Peter J. Knaggs. Type inference in stack based languages. *Formal Aspects of Computing*, 5(4):289–298, 1993.

[Smi92] Norman Smith. *Write Your Own Programming Language Using C++*. Wordware Publishing, 1992. ISBN 1-55622-264-5.

[Sto88] Bill Stoddart. Specification & optimisation. In *euroFORML'88 Conference Proceedings*, pages 147–165, MPE Ltd, 133 Hill Lane, Southampton SO1 5AF, UK., September 1988. Forth Interest Group.

[Sun95] Sun Microsystems. *The Java Virtual Machine Specification*, 1.0 beta edition, August 1995.

[Tev89] Adin Tevet. Symbolic stack addressing. *Journal of Forth Application and Research*, 5(3):365–379, 1989.

[TvSKS83] Andrew S. Tanenbaum, Hans van Staveren, E. G. Keizer, and Johan W. Stevenson. A practical tool kit for making portable compilers. *Communications of the ACM*, 26(9):654–660, September 1983.

[Ung87] David Ungar. *The Design and Evaluation of a High-Performance Smalltalk System.* MIT Press, 1987.

Index