Stack Caching for Interpreters

M. Anton Ertl
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
`anton@mips.complang.tuwien.ac.at`
Tel.: (+43-1) 58801 4459
Fax.: (+43-1) 505 78 38

Abstract

An interpreter can spend a significant part of its execution time on accessing arguments of virtual machine instructions. This paper explores two methods to reduce this overhead for virtual stack machines by caching top-of-stack values in (real machine) registers. The *dynamic method* is based on having, for every possible state of the cache, one specialized version of the whole interpreter; the execution of an instruction usually changes the state of the cache and the next instruction is executed in the version corresponding to the new state. In the *static method* a state machine that keeps track of the cache state is added to the compiler. Common instructions exist in specialized versions for several states, but it is not necessary to have a version of every instruction for every cache state. Stack manipulation instructions are optimized away.

1 Introduction

Interpreters are often used for implementing programming languages. Their major advantages 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.

The present paper deals with these issues. Section 2 discusses general efficency issues; then we concentrate on a particular aspect of efficency: accessing arguments of virtual machine instructions. Our solution uses a stack machine that caches stack values in registers (Section 3). We present two methods for implementing this idea: either the interpreter (Section 4) or the compiler (Section 5) keeps track of the cache state. Finally, empirical results are presented (Section 6).

The main original contributions of this paper are the compiler-based static stack caching technique, the discussion of different stack cache organizations, and the empirical evaluation.

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.

Some examples are written in MIPS assembly: register n is denoted by `$`n, the destination operand of an instruction is usually the leftmost register, and comments start with `#`.

2 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.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
SIGPLAN '95 Conference on Programming Language Design and Implementation, pages 315-327

```
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 1: Direct threading in C using tail calls

```
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: Instruction dispatch using `switch`

2.1 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 fetching that address and jumping to the routine. 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. 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) or calls (Fig. 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

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

Inst *ip;
int *sp;

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

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

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

Figure 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 4: Direct threading in MIPS assembly

I call it direct call threading. Figure 4, 5 and 6 show MIPS assembly code for the three techniques (direct call threading needed a little source code twisting to get reasonable scheduling). Fig. 7 shows the overhead of these techniques in cycles on two processors, 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 pointers, have 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 Fig. 8). If portability to machines without `gcc` is a concern, it is easy to

```
$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 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 6: Direct call threading in assembly

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. 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).

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

Figure 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 8: Direct threading using GNU C's "labels as values"

2.2 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).

2.3 Accessing arguments

In the hardware area, the contest between stack and register architectures has been decided for register machines.[1] However, for interpretive implementations

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

```
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 9: Add in a register architecture (without instruction dispatch)

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

Figure 10: Unfolded add (r1 and r2 into r3)

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[2] and move instructions necessary in register architectures are much more time consuming than in hardware, since 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. 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

[2]If there are more values than the compiler can keep in registers, some values have to be stored into memory and loaded back later. This is called *spilling*.

```
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 11: Add in a simple stack implementation

```
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 12: Add, the top of stack is kept in 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 registers can then be accessed directly, and therefore be kept in real machine registers, if there are enough[3]. Fig. 10 shows one 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. 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.

If there are enough registers, the number of operand fetches and stores can be reduced by keeping n top-of-stack values in registers (see Fig. 12). This is not always beneficial; if an instruction takes x items from the stack and stores y items to the stack, 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.

[3]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.

- is as fast as keeping $n-1$ items in the other cases.

This holds for all machines where loads, stores, and moves cost more than zero cycles. Moreover, machines that can exploit a high amount of instruction-level parallelism can profit from the prefetching effect of keeping more items in registers. On a related note, keeping one item in a register also speeds up floating-point and other long-latency instructions, where the store back to the stack would expose the latency.

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 6).

3 Stack caching

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 cache.

This requires different implementations of an instruction for different cache states. Every allowed mapping of stack items to machine registers constitutes a cache state.

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 one state to another. Fig. 13 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.

3.1 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 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. 13). This means that the stack pointer need not be updated

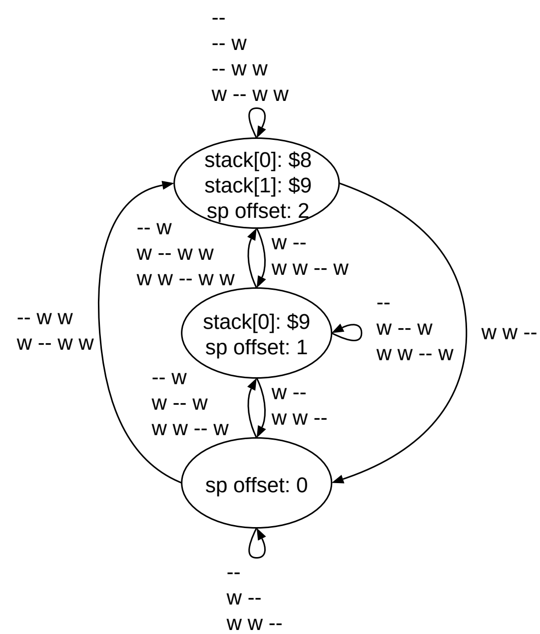

Figure 13: A simple cache state machine (transitions are marked with stack effects. `w w -- w` represents an instruction that takes two word-sized items from the stack and puts one result back on the stack (e.g., add).)

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

Figure 14: Add in stack caching (starting in the full state of the three-state machine)

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. 14).

3.2 Minimal organization

As a minimum, there should be one state for every number of stack items in registers (as in Fig. 13). 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 similarly. This arrangement of states avoids the need to move stack items around on the bottom of the cache whenever something on the top changes.

3.3 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,

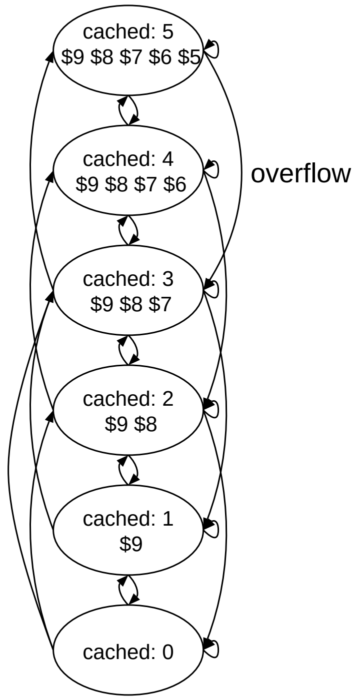

Figure 15: Overflow transition in a minimal organization (the top-of-stack is rightmost, `$9` contains the deepest cached item)

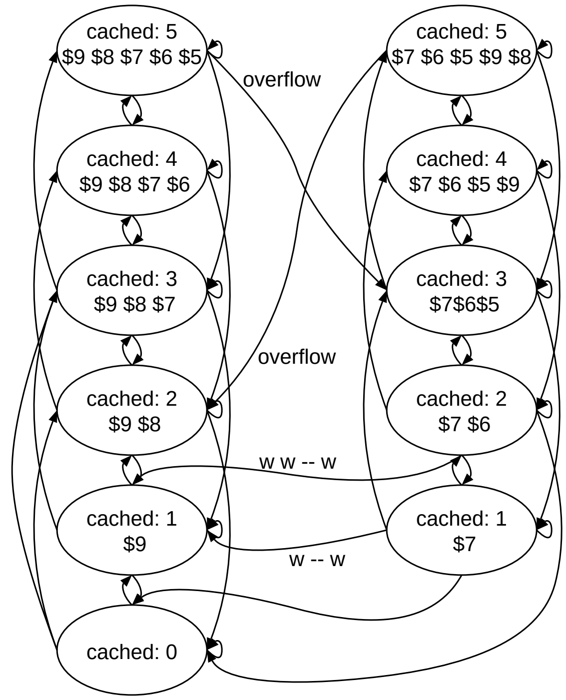

Figure 16: Avoiding moves with additional states

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 there are also other costs associated with overflowing: the movement of stack items to other registers and the updating of the stack pointer. 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. 15), in order to reduce the number of overflows (even though this increases the number of underflows a bit). Which state is the best, is probably best determined empirically. While there are theoretical results [HS85], they are based on a random walk model, where pushes and pops occur equally likely irrespective of previous events. It is not clear that this model describes the behaviour of real programs, and our empirical results indicate that it does not (see Section 6).

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 can be dependent on the executed instruction.

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 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. 16).

3.4 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 stack items to registers where no register occurs twice. For duplicating instructions (e.g., `dup` and `over`), the extreme form results in an infinite number of cache states, since an unlimited number of such instructions causes an equally unlimited number of stack items to reside in

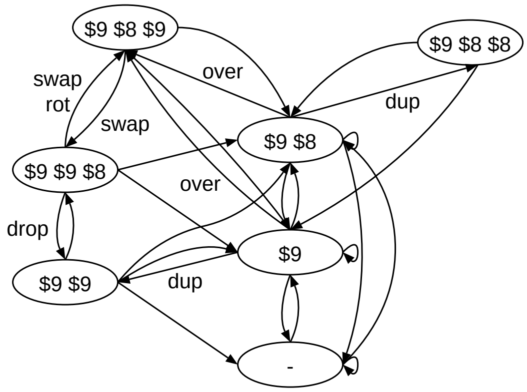

Figure 17: A cache organization where one duplication is allowed (`dup` duplicates the top of stack, `over` duplicates the second item, `swap` swaps the two top items, `rot` rotates the third item to the top, `drop` pops the top item)

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 17 shows a two register cache organization where one duplication is allowed.

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 again be optimized by introducing additional states.

3.5 Reducing the number of states

In practice finiteness is not enough, there are also other limits to the number of states. Figure 18 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; “arbitrary shuffles” optimizes shuffle instructions in a similar way, “$n+1$ 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., a half-full state in the above-mentioned overflow case, because it 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.[4] Or, even better, if the future is known, 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.

3.6 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.

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 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.

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

registers	1	2	3	4	5	6	7	8	n
"minimal"	2	3	4	5	6	7	8	9	$n+1$
overflow move opt.	2	5	10	17	26	37	50	65	n^2+1
arbitrary shuffles	2	5	16	65	326	1,957	13,700	109,601	$\sum_{i=0}^{n} n!/i!$
$n+1$ stack items	3	15	121	1,356	19,531	335,923	6,725,601	153,391,689	$\sum_{i=0}^{n+1} n^i$
one duplication	3	7	14	25	41	63	92	129	$n(n+1)(n+2)/6+n+1$
two stacks	3	6	9	12	15	18	21	24	$3n$

Figure 18: The number of cache states

```
$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 instrcution 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 19: Add in dynamic stack caching with table lookup

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. 19). 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 rest has been optimized away (as in the add in state 2 in Fig. 19). 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 DecStation 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. 19 is faster (on both the R3000 and R4000) than the add in Fig. 12 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[5] caching, or, if there are few registers for caching, one duplication, to make better use of them. Eliminating the moves of stack manipulation instructions does not pay in many cases: 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.

5 Static stack caching

In static stack caching the compiler keeps track of the state of the cache and generates the code accordingly.

This approach offers several big advantages over dynamic stack caching:

- There is no need for a special instruction dispatch

[5] Languages with user-visible stacks (e.g., Forth) have a separate stack for storing the return addresses of calls.

routine and its possible performance disadvantages, direct threading can be used.

- Stack manipulations can be optimized away completely, i.e., not even an instruction dispatch is executed. The compiler just notes the state transition.
- 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 generate 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.

Prog.	Instr.	loads	updates	rloads	rupdates	calls
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
cross	4,914,610	0.74	0.51	0.19	0.33	0.14

Figure 20: 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, and calls per instruction.

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 specialization 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.

6 Empirical results

We instrumented a Forth system to collect data about the behaviour of various stack caching organizations.[6] 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 (*cross*). Figure 20 shows the number of loads from the stack (same as the number of stores to the stack), stack pointer updates and ex-

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

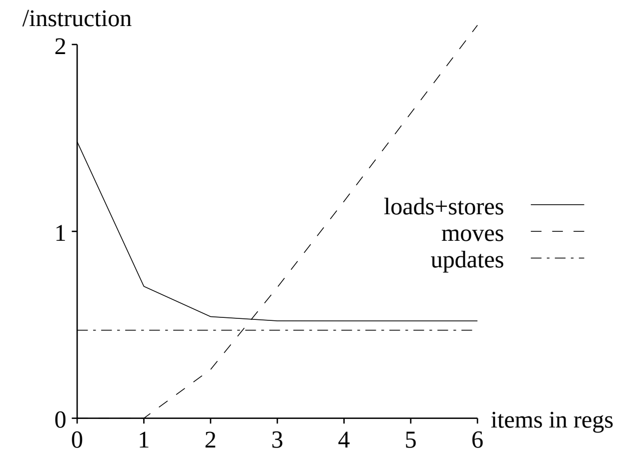

Figure 21: Keeping a constant number of items in registers: Memory accesses, moves, and stack pointer updates per instruction vs. number of items kept in registers

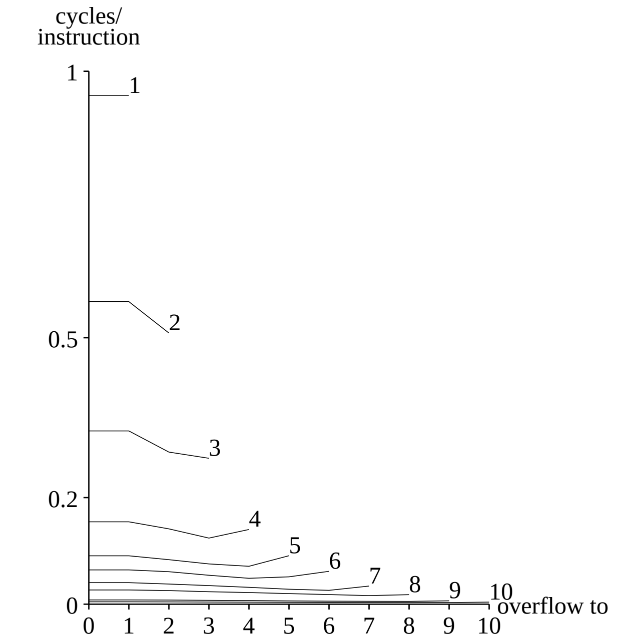

Figure 22: Dynamic Stack Caching: Argument access overhead in cycles/instruction of minimal organizations with different numbers of registers vs. overflow followup state

ecuted (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) or pops (on returns); therefore, always keeping one return stack item in a register has virtually no effect.

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 sum of all programs.

First, we measured the effect of keeping a **constant number of stack items** in registers (see Fig. 21). It is easy to see that keeping one item in a register is best (see also Fig. 26): 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 (R3000) keeping one item in a register causes a speedup of 11% for *prims2x* and 7% for *cross*.[7]

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. 22 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 overhead 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.[8] Another interesting result is that the optimal overflow followup states are rather full, while we expected them to be about half-full, based on the results in [HS85]. In this light, the underflow behaviour we used is probably close to optimal.

Figure 23 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. 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 (note that underflows are usually one item

[7]The other programs run too fast to produce exact timings. Explicit register declarations were used to keep gcc from spilling important registers.

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

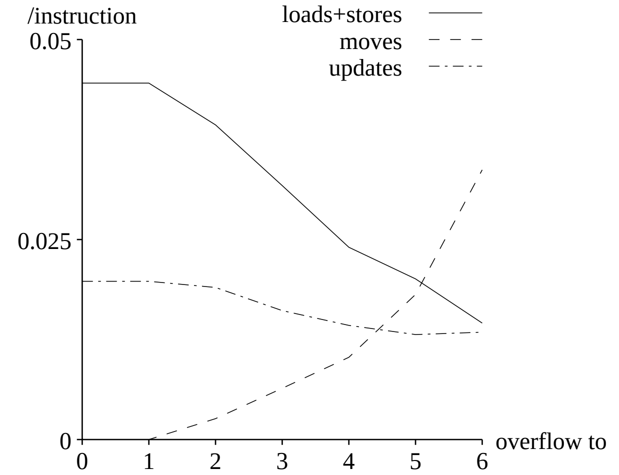

Figure 23: Dynamic Stack Caching: Memory accesses, moves, and stack pointer updates per instruction of organizations with 6 registers for varying overflow followup states

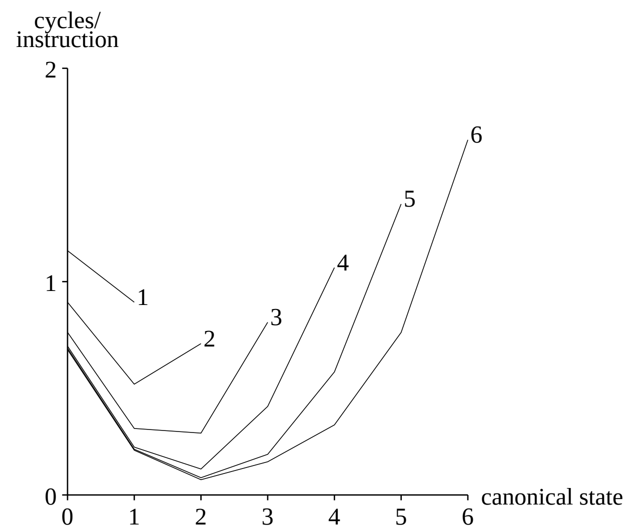

Figure 24: Static Stack Caching: Argument access overhead in cycles per (original) instruction of various organizations vs. canonical state

at a time, while overflows typically spill several items at a time). However, this does not hold for higher numbers of registers.

For **static stack caching** we 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; we tried all the states of the minimal organization as canonical state (which also served as overflow followup state). In Fig. 24 the lines represent the performance of cache

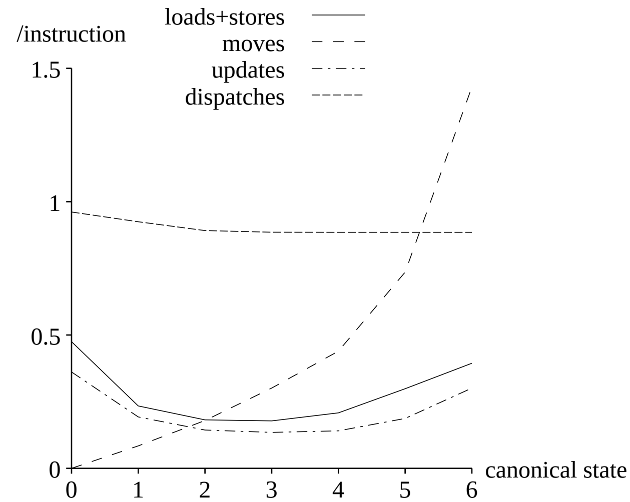

Figure 25: Static Stack Caching: Memory accesses, moves, stack pointer updates and instruction dispatches per (original) instruction of organizations with 6 registers for varying canonical states

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. 25). Increasing the 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. 24 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 we 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.

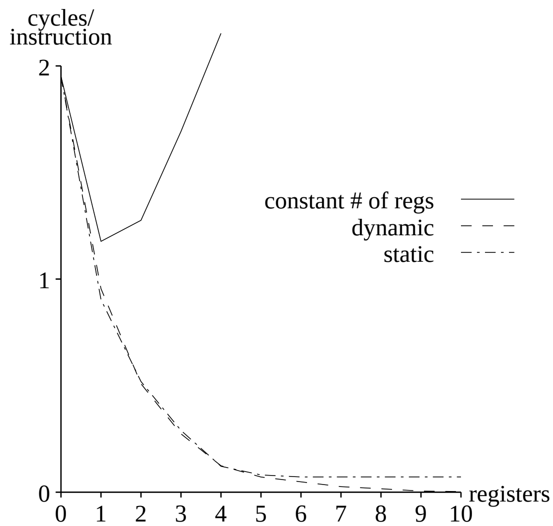

Figure 26: Comparison of the approaches: argument access overhead in cycles/instruction vs. number of registers used

We 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) makes us believe that vast reductions are possible with little negative impact on the execution time.

Comparison. Figure 26 compares the three approaches. For dynamic and static stack caching the best of the evaluated organizations for a specific 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 we 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).

Finally, we took a closer look at the empirical data in order to see how well the **random walk model** of [HS85] describes the behaviour of our programs: In *cross* and *compile*, the number of overflows is not reduced by changing the overflow followup state of the 10-register cache from a state with seven items in registers (state 7) to a state with fewer items in registers. This means that, after none of the overflows (and there were 1110 in these programs with these cache organizations), more than three more stack items were pushed before an underflow happened. In other words, there's a very strong tendency to go down after going up. By contrast, the random walk model assumes that the behaviour is independent of previous behaviour. It predicts that (whether there was an overflow or not) in state 7 of a 10-register cache there is a higher probability of overflow than underflow. Obviously, the random walk model does not describe the behaviour of these programs well. What about the other programs? In *prims2x*, the overflows do not decrease for overflow followup states below state 5, giving essentially the same picture. Only in *gray* this symptom does not appear in the 10-register cache. This is probably due to the fact that *gray* performs a graph walk using recursion. Still, in less than 10 of the 279 overflows to state 5, *gray* overflows another time before underflowing. Until better models are available, stack-based designs have to be evaluated empirically.

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, 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.

8 Conclusion

Apart from optimizing instruction dispatch and increasing the semantic content of the instructions, another factor determines the performance of an interpreter: accessing the arguments of the instructions. For interpreters conventional register architectures do not enjoy the same advantages as on hardware implementations. Their disadvantages are compiler complexity, slowness and/or big interpreters.

The performance of virtual stack machines can be improved by caching stack items in registers. 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.

Acknowledgements

Konrad Schwarz, Robert Bernecky, Andi Krall, Franz Puntigam and Manfred Brockhaus, Marcel Hendrix, Ulrich Neumerkel and the referees provided valuable comments on earlier version of this paper. Marty Fraeman, John Hayes and Chris Bailey discussed about program behaviour and their experiences with the random walk model with me.

References

[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.

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

[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.

[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`.

[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.

[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 interpreters: A management algorithm and its analysis. In *International Symposium on Computer Archictecture (ISCA)*, pages 48–54, 1985.

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

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

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

[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.