Stack Caching for Interpreters

M. Anton Ertl

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

Abstract. An interpreter for a virtual stack machine can spend a significant part of its execution time fetching values from and storing values to the stack. This paper explores two methods to reduce this overhead by caching top-of-stack values in registers. The dynamic method is based on having one version of the whole interpreter for every possible state of the cache; the execution of a primitive usually changes the state of the cache and the next primitive 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 primitives exist in versions for several states, but it is not necessary to have a version of every primitive for every cache state. The compiler generates glue code, if necessary, and compiles the version of the primitive appropriate for the cache state. Stack manipulation primitives are usually optimized away.

1 Introduction

Interpreters are often used for programming language implementation. The major advantages over compilation to native code are simplicity and portability. The 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, they are even used in special implementations of traditionally compiled languages like C, and probably the majority of special-purpose language implementations are interpreters.

In the last years many questions about interpreters have been asked in the Usenet newsgroup `comp.compilers`. Efficiency was a major concern; another question that came up several times 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 the efficency question, the question of accessing arguments of virtual machine instructions. Our answer is to use a stack machine that caches a variable amount of stack values in registers (Section 3). We present two methods for implementing this idea: Either the interpreter keeps track of the cache state (Section 4, or the compiler does it (Section 5).

A note on teminology: 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.

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 parts constitute the interpreter overhead.

2.1 NEXT

```
lw   $2,0($4) #$4=ip
addu $4,$4,4
j    $2
#nop          #branch delay slot
```

Fig. 1. Direct threading in MIPS assembly

The most efficient method for fetching, decoding, and starting the next primitive is still direct threading [Bel73]. 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 shows how direct threading would be implemented in C using

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: EuroForth '94 Conference Proceedings, pages 3--12

```c
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 /* ... */ };
```

Fig. 2. Direct threading in C using tail calls

```c
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;
    }
}
```

Fig. 3. Instruction dispatch using `switch`

```
$L2: #for (;;)
 lw    $3,0($6) #$6=ip
 #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 #$7 contains $L13
 lw    $2,0($2)
 #nop
 j     $2
 #nop
 ...

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

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

Fig. 4. Switch dispatch in assembly

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

Inst *ip;
int *sp;

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

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

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

Fig. 5. Direct call threading

```
add:
 ...
 j    $31

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

Fig. 6. Direct call threading in assembly

tail-calls). Two methods are usually used in C: a giant switch (Fig. 3) or calls (Fig. 5). In the first method the whole interpreter, including the implementations of the instructions, must be in one function. In the second method every primitive 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 1, 4

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

Fig. 7. Cycles needed for instruction dispatch

and 6 show MIPS assembly[1] 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 (all delay slots filled). 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.

```
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++;
}
```

Fig. 8. Direct threading using GNU C's "labels as values"

Fortunately, there is a widely-available language with first-class labels: GNU C (version 2.x); so we can implement direct threading portably (see Fig. 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 as well as 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 control flow of the interpreted program changes (unless there are delayed branches in the virtual machine).

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 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.[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 eliminates one of the advantages of using an interpreter, namely simplicity.

Moreover, in an interpreter the spill and move instructions necessary in register arechitectures are much more time consuming than in hardware, since each instruction also has to execute a NEXT. This is not balanced by the fact that the other instructions also have to perform NEXTs, since the other instructions usually have higher semantic content. E.g., for a direct threaded implementation on the R4000 a spill or move instruction is (at least) 7 times more expensive than in native code, whereas, e.g., an instruction for computing the maximum of two numbers is not even twice as expensive as in native code.

[1] In MIPS assembly, register n is denoted by $\$n$, and the destination operand of an instructions is usually the leftmost register.

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

```
lw   $3,0($6) #$6=ip
lw   $2,4($6)
lw   $4,8($6)
addu $3,$7,$3 #$7=reg. array start
addu $2,$7,$2
lw   $2,0($2)
lw   $3,0($3)
addu $4,$7,$4
addu $2,$2,$3
sw   $2,0($4)
```

Fig. 9. Add in a register architecture (without NEXT)

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 NEXT on the MIPS architecture (10 cycles on R3000).

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

Fig. 10. Unfolded add (r1 and r2 into r3)

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 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.

```
lw   $2,0($5) #$5=sp
lw   $3,4($5)
addu $2,$2,$3
sw   $2,4($5)
addu $5,$5,4
```

Fig. 11. Add in a simple stack implementation

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.

```
lw   $2,4($5) #$5=sp
addu $5,$5,4
addu $6,$6,$2 ;$6=tos
```

Fig. 12. Add, the top of stack is kept in a register

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.

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 probably not a good idea, because from the top ten heavily-used instructions three (16% of all

[3] However, the availability of registers should not be taken for granted even on the register-rich RISCs. E.g., when I tried to keep the top of stack (of Forths 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.

dynamically executed instructions) become slower, and only one (5% of the executed instructions) becomes faster. One (2.6%) may profit from prefetching.

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.

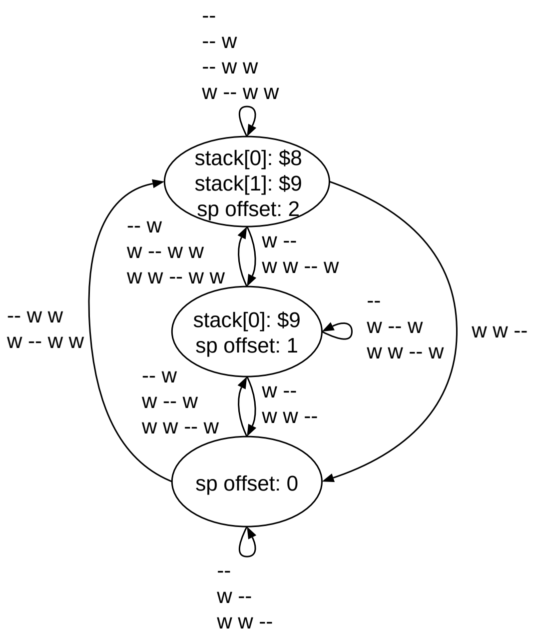

Fig. 13. A simple cache state machine

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 interesting optimization problem that we leave for future work. Here we present just a few insights.

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 in instruction implementations that can access all stack items in registers, i.e., hopefully most of the time.

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

Fig. 14. Add in stack caching (full state of the three-state machine)

Stack caching with stack pointer update minimization leads to code that is as good as that of the unfolded register machine (see Fig. 14).

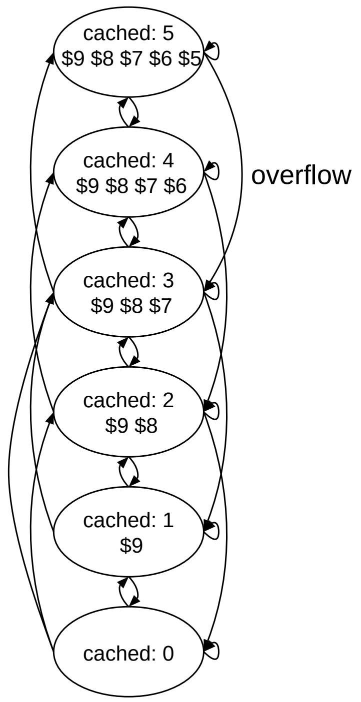

Fig. 15. Overflow transition in a minimal organization

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. 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: On many processors a store to cached memory costs as much as a move, therefore on overflow the transition to any state costs the same amount. The best choice is usually a slightly more than half-full state (see Fig. 15): this makes cache over- or underflows in the near future pretty unlikely.

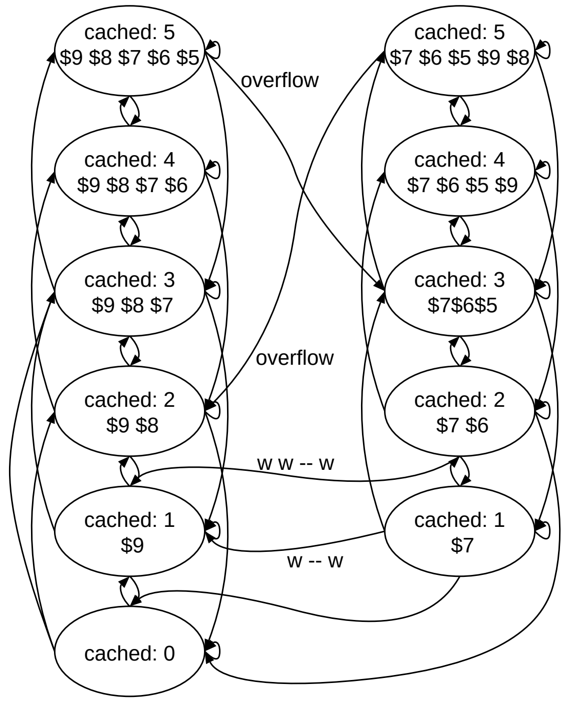

Fig. 16. Avoiding moves with additional states

Another solution to the movement problem 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. But 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).

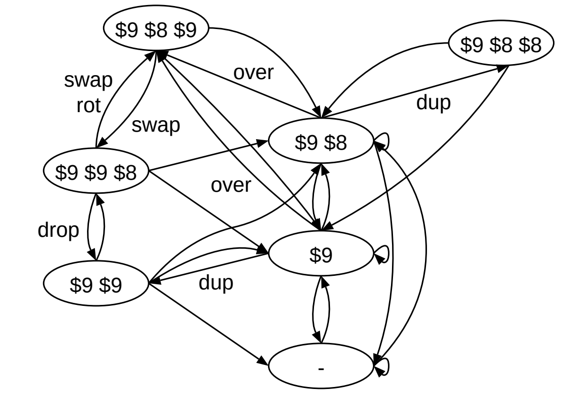

Fig. 17. A cache organization where one duplication is allowed

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 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). This is a good solution for Forths floating-point stack on machines that have a separate floating-point register set (nearly all current machines). They can also be treated in a unified manner, sharing the same set of registers. This is the solution of choice for Forths data and return stacks. Moves between the stacks can again be optimized by introducing additional 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

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	$n(n+1)(n+2)/6+n+1$
return stack	3	6	9	12	15	18	21	24	$3n$

Fig. 18. The number of cache states

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. "Return stack" is the "minimal" organization, combined with caching up to two return stack items 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 this case 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, 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 processor 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.

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. 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 pipeline bubbles.

4 Dynamic stack caching

Dynamic stack caching is a pure run-time method, i.e., the interpreter maintains the state of the cache and the compiler need not be aware of it. 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 NEXT 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 path length of the NEXT 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 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. The problem with this approach is that no portable language I know supports placing routines at specific points in

[4] 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) #$6=sp
 lw   $3,4($6)
 lw   $2,0($5) #$5=ip
 addu $6,$6,8
 lw   $2,4($2) #next state: 1
 addu $5,$5,4
 j    $2
 addu $4,$4,$3

$L3: #add in state 1: tos in $4
 lw   $2,0($6)
 lw   $3,0($5)
 addu $6,$6,4
 lw   $3,4($3) #next state: 1
 addu $5,$5,4
 j    $3
 addu $4,$4,$2

$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
 j    $2
 addu $5,$5,4
```

Fig. 19. Add in dynamic stack caching with table lookup

memory; Even worse, even some assemblers do not support it (e.g., the DecStation assembler).

If NEXT becomes more expensive, dynamic stack caching is probably not worth the trouble.

Since the whole interpreter has to be replicated for every state, only state machines with a few dozen states or less (depending on the size of the interpreter and the (real machine) instruction cache) are practicable. 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. Eliminating the moves of stack manipulation instructions does not pay in many cases anyway: The NEXT has to be performed anyway, and the moves can often be done in parallel, i.e., in the delay slots. In Forth return stack caching would be very profitable, given the high frequency of calls and returns. A nice optimization is possible here: The 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 register.

Since the state of the cache is represented in only one value, i.e., the program counter of the processor, it is not possible treat two caches (e.g., for data and 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 having 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 NEXT routine and its possible performance disadvantages, direct threading can be used.
- There is no need to replicate the whole interpreter for every state: First, the implementation of the same instruction in many states can be the same, i.e., when the arguments of the instruction are accessed in the same registers, but some other stack items reside in different registers etc. (in dynamic stack caching they would have different NEXTs for continuing in different states); second, implementations of rarely used instruction 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.
- Stack manipulations can be optimized away completely, i.e., not even a NEXT is executed. The compiler just notes the state transition.
- 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 insufficent 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 definitions 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.[5]

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

[5] This implies that there are several versions of the call instruction, so the conventional Forth way calling cannot be taken and an explicit call instruction with an inline argument is needed.

branch target) the code is in a specific state. The transition into this state can be performed by the branch instructions; for branch targets the transitions have to be performed by additional instructions generated just in front of the target. A slightly more complex solution is to generate no code before branch targets; the transition to the state at the branch target must be performed by the branch. This avoids generating additional instructions.

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 definition the register is filled with the return address. This provides the leaf procedure optimization of conventional languages on RISCs. After a call the return stack cache is empty (the value has just been used up in the return from the call).

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. 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 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 a big 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 [Koo89, HFWZ87]. For interpreters, [DV90] proposed dynamic stack caching with a "minimal" cache organization. However, they do not analyse the available options as it is done in Section 3. In particular, they do not optimize stack pointer updates away (this may also be due to their use of the 8086 for their examples), and their successor state for overflow is the full state. They report speedups (probably over an implementation that does not keep any part of the stack in registers, probably running the sieve benchmark) 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. They also report a reduction of stack referencences in Forth of 54% for one register, 82% for two registers (this difference results in a 5% speedup on the 8086) and 93% for four registers. For M-code, the reductions are 56% for one register and 100% for three registers. They do not report the number of additional moves.

7 Conclusion

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

The performance of 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 much of the speed advantage of caching. In static caching the compiler keeps track of the cache state. This allows using organizations with more states, using 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.

I am currently working on implementing these ideas in a Forth interpreter generator [Ert93], which can then be used for getting empirical results for various organizations on several processors.

Acknowledgements

Konrad Schwarz provided valuable comments on this paper.

References

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

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

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

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

[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.

[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.