Combining Stack Caching with Dynamic Superinstructions

M. Anton Ertl*
TU Wien

David Gregg
Trinity College, Dublin

Abstract

Dynamic superinstructions eliminate most of the interpreter dispatch overhead. This results in a higher proportion of interpreter time spent in stack accesses (on a stack-based virtual machine). Stack caching reduces the stack access overhead. Each of these optimizations provides more speedup, if the other one is applied, too. Combining these optimizations also opens additional opportunities: we can insert stack state transitions without dispatch cost; this reduces the number of necessary VM instruction instances significantly. A shortest-path search can find the optimal sequence of state transitions and VM instructions. In this paper we describe an implementation of static stack caching employing these ideas. We also represent empirical results for our implementation, resulting in a speedup of up to 58% over a version that keeps one value in registers all the time.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors—*Interpreters*

General Terms

Languages, Performance, Experimentation

Keywords

Interpreter, virtual machine, stack caching, superinstruction, shortest-path algorithm

*Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

1 Introduction

The execution of a virtual machine (VM) instruction consists of three parts

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

Stack caching [Ert95] reduces the argument access overhead for stack-based VM interpreters; Section 2.1 gives an overview of stack caching.

Dynamic superinstructions [RS96, PR98, EG03] reduce the dispatch overhead by eliminating most dispatches (and, on some processors, by improving branch prediction [EG03]); Section 2.2 gives an overview of dynamic superinstructions.

Since these optimizations work on different parts of the interpreter and can therefore be applied independently, they support each other: each provides more speedup if the other optimization is applied, too.

However, combining these optimizations provides an additional benefit: Dynamic superinstructions allow transitions between stack states to be introduced without any dispatch overhead, allowing more flexibility in choosing VM instruction instances (see Section 3.2).

We also discuss the use and the benefits of using a shortest-path algorithm for selecting the optimal instruction/state combinations and state transitions (Section 3.3), and give an overview of how the code for stack caching is generated by the virtual machine generator vmgen (Section 4.1) and how these pieces come together in a working system (Section 4.2). Finally, we present empirical results about the speedups achieved from stack caching (Section 5).

The contributions in this work are that we present the first implementation of static stack caching, explore the combination of static stack caching with dynamic superinstructions, and the opportunities of this combination (dispatch-free state transitions, fewer instruction/state combinations), discuss the use of a shortest-path algorithm in depth, and present empirical results for an implementation containing all these features.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
SIGPLAN 2004 Workshop on Interpreters, Virtual Machines, and Emulators (IVME '04) Proceedings, pages 7-14

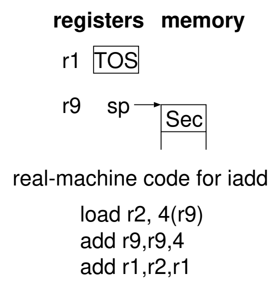

Figure 1. A simple and frequently-used stack representation, and the implementation of the JVM `iadd`.

1.1 Why stack-based VMs?

Stack-based VMs have less argument access overhead than register architecture VMs, for the following reasons:

- Less VM instruction decoding overhead: The location of many of the arguments of a VM instruction is implicit (top-of-stack), whereas a stack-based VM has to fetch and decode the register numbers.
- Explicitly-addressed VM registers have to be held in real-machine memory, whereas the top-of-stack item(s) can be held in real-machine registers without much complication.

These reasons outweigh the stack pointer update overhead that a stack machine has (which can be reduced with multiple-state stack caching).

On the other hand, register VMs perform fewer dispatches [DBC+03].

So, whether a stack or a register VM is faster depends on the dispatch cost. With dynamic superinstructions, the dispatch cost within a basic block is zero (if you do not keep the superfluous VM instruction slots [PR98]) or very small (if you keep the slots [EG03]). So, in this environment, we expect stack machines to be faster, and multiple-state stack caching makes them even faster.

2 Previous work

2.1 Stack Caching

This section gives an overview of stack caching [Ert95].

The simplest representation of the stack keeps all stack items in memory, and maintains a stack pointer to the top-of-stack. An easy, frequently-used and worthwhile improvement over that is to keep the top-of-stack in a register, and keep the rest of the stack in memory (Fig. 1). This improvement has been called single-state stack caching [Ert95]; it approximately halves the number of real-machine loads and stores to stack items.

If we use several different stack representations, we can reduce the overhead for accessing the stack even more. E.g., consider the stack representations S1 and S2 in Fig. 2: S1 is the same stack representation as in Fig. 1, i.e., it keeps one stack item in registers, the rest in memory, and the stack pointer points to the first item that's in memory; S2 differs from this by keeping two stack items in registers.

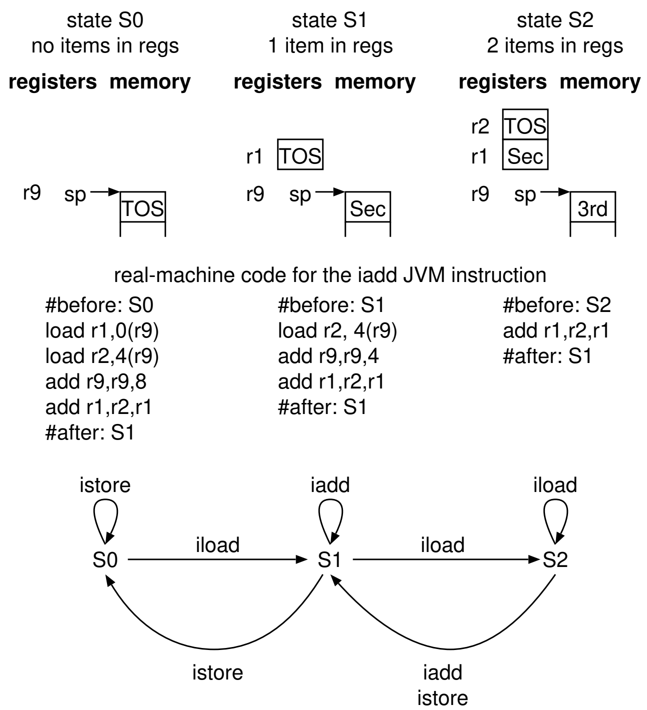

Figure 2. Three stack representations (states), implementations of the JVM `iadd` for these states, and a state machine with transitions for three JVM instructions.

If we use only one representation, using S2 is about as good as S1; e.g., `iadd` also needs two adds and a load to implement. However, if we let the `iadd` start in S2 and end in S1, the implementation becomes much simpler, consisting just of a single register-to-register add.

The registers we use constitute our *stack cache*, we call the stack representations used the stack-cache *states*.

Of course, the next VM instruction has to start in the state where the last one ended. So, in the simplest case, we need one *instance* of each VM instruction for every stack state we use. Figure 2 shows the three implementations of the three instances of `iadd` for the three states (without dispatch).

As a result, the interpreter becomes a finite-state machine, with VM instructions as transitions between the stack states. Figure 2 shows the transitions that work best for the JVM instructions `iadd`, `iload`, and `istore`.

Consider a common sequence like

```
iload b
iload c
iadd
istore a
```

(e.g., resulting from the Java statement `a=b+c;`). Starting in S0, the transitions shown in Fig. 2 result in executing code that accesses all implicit stack items in real-machine registers (not in memory), and does not perform any stack pointer updates. In general, multiple-state stack caching can eliminate most loads and stores to stack

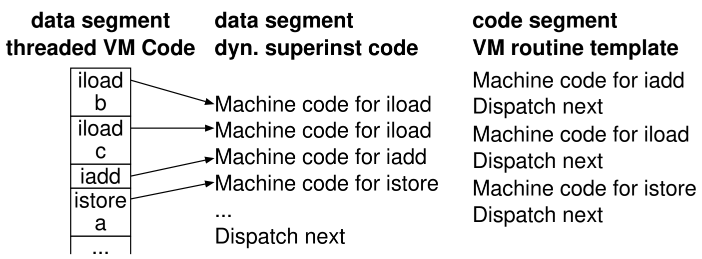

Figure 3. A dynamic superinstruction for a simple JVM sequence

items and most stack pointer updates (depending on the number of registers and states available).

The system has to keep track of the stack state and to dispatch the right instance of the instruction. There are two ways to achieve this:

- In dynamic stack caching the interpreter keeps track of the state, typically by dispatching through a different dispatch table (one table per state). This can be done without executing additional instructions in a byte-code interpreter (which uses tables anyway), but it cannot be used in a threaded code interpreter [Bel73], because threaded-code dispatch does not perform a table-lookup.[1]
- In static stack caching the front end (the part of the system that produces the VM code) keeps track of the state, and records the VM instruction instance that the interpreter has to execute; therefore the interpreter dispatch does not need to deal with states. This is a good approach for threaded-code interpreters, but it may require introducing additional state transition VM instructions before control-flow joins and (depending on the sophistication of the front end) before other control-flow.

2.2 Dynamic Superinstructions

This section gives a simplified overview of dynamic superinstructions [RS96, PR98, EG03].

With dynamic superinstructions, the machine code for the VM instructions is concatenated together (see Fig. 3), eliding the dispatch code for all VM instructions except for taken VM branches (where falling through (the result of eliding the dispatch code) would produce the wrong result).

Dynamic superinstructions are created by the front end of the interpretive system:[2] Whenever a VM instruction is compiled, the front end copies the machine code for the instruction from a template to the end of the current dynamic superinstruction; if the VM instruction is a VM branch, it also copies the dispatch code. In addition to the machine code, the front end also produces VM code; the VM instructions are represented by pointers into the machine code of the superinstruction (for use with threaded-code dispatch).

As a result, most of the dispatches are eliminated, and the rest have a much better prediction accuracy on CPUs with branch target buffers, thus eliminating most of the dispatch overhead. In particular, there is no dispatch overhead in straight-line code.

There is one catch, however: Not all VM instruction implementations work correctly when executed in another place. E.g., if a piece of code contains a relative address for something outside the piece of code (e.g., a function call on the IA32 architecture), it is not relocatable for our purposes. The way to deal with this problem is to append a dispatch to the end of the current superinstruction and let the VM instruction slot point to the original template code for this instruction. In this way the CPU will execute the superinstruction, then dispatch to the non-relocatable instruction template (which works correctly in its original place), and then dispatch to the next superinstruction or template.

3 Combining stack caching and dynamic superinstructions

3.1 Static or dynamic stack caching?

The combination of these techniques is fairly straightforward, but dynamic superinstructions have some effects on the design decisions for stack caching.

First, dynamic stack caching is pretty hard to implement in this context, as it provides a number of disadvantages: For each state, an instance of the superinstruction would have to be created; the additional code size could increase the number of I-cache misses significantly, especially since there is no obvious and easy way to arrange the additional code in a way that preserves spatial locality. For every VM branch target, dynamic stack caching would require a variant for each state; this is easiest to achieve by ending dynamic superinstructions before all VM branch targets. The dispatch would have to deal with keeping track of the state, which causes problems with threaded code dispatch, as discussed above. In addition, the front end has to keep track of the state while generating the instances of the dynamic superinstructions, eliminating one of the few advantages of dynamic stack caching.

In contrast, static stack caching is helped by dynamic superinstructions: Without static superinstructions, the state-transition instructions that may be necessary before control flow joins (or, for less sophisticated front ends, at each basic block boundary) require an additional dispatch; dynamic superinstructions eliminate this dispatch, and even the VM instruction slot associated with it (making it easier to drop in static stack caching into an existing interpretive system).

3.2 Using state transitions

One problem with stack caching is that it multiplies the number of VM instruction implementations to be compiled by the number of states.[3] Since the time and memory required by gcc to compile

[1] One way to deal with this would be to expand the threaded code to contain several threaded-code slots for each VM instruction [SC99], one for each state, but this would result in a very large threaded code size.

[2] I.e., dynamic superinstruction creation and static stack caching instance selection happen at the same time; they are called the way they are to contrast them against other superinstruction and stack caching variants: static superinstructions are created earlier, at interpreter build time; dynamic stack caching selects the VM instruction instance later, at run-time.

[3] If an implementation is the tail of another implementation, they can share code [PWL04, Gri01]; however, for more complex stack cache organizations this is not always possible and it complicates the generation of the interpreter.

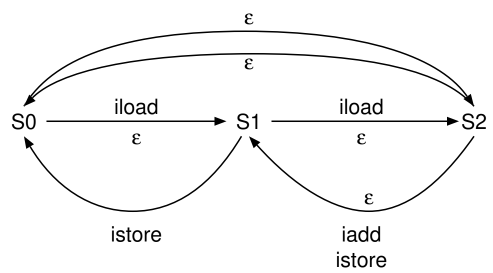

Figure 4. Some VM instruction instances can be replaced by sequences of state transitions (ε) and other instances of the same VM instruction.

the interpreter engine (all in one function) increases about quadratically with the number of VM instruction instances, there is a practical limit of about 1000 VM instruction instances. So, how can we reduce the number of VM instruction instances?

First we can leave rarely-used VM instruction instances away, keeping just one instance per VM instruction. If the front end has to compile this VM instruction, but is in the wrong state, it can insert transition code for getting from the current state to the desired state; on the VM level these transitions are no-ops. Thanks to dynamic superinstructions these transitions usually do not cost an additional dispatch.

Another opportunity for reducing combinations reveals itself if you look at the instances of `iadd` in Fig. 2: The code for the S2 instance is a suffix of the code for the S0 and S1 instances. The other code in the S0 and S1 instances is just state transition code. So we do not need the S0 and S1 instances, because we can replace them by a state transition (S0→S2 and S1→S2 respectively) followed by the S2 instance of `iadd`.

In general, for a simple stack cache organization like that in Fig. 2, if a VM instruction takes *n* items from the stack, there is no need for VM instruction instances that start in stack states with $< n$ items in registers; these can be replaced by the sequence of a state transition followed by the the VM instruction instance starting with *n* items in registers.

Also, if a VM instruction takes no items from the stack and pushes *n* items on the stack, it is not necessary to have VM instruction instances where less than *n* registers are empty at the start.

Applying these changes to our example results in the state machine in Fig. 4.

3.3 Optimal code generation

Static stack caching requires that the front end keeps track of the state and selects the right instance of the VM instruction for the current state, possibly inserting state transitions. We call this task *code generation*.

If there is exactly one instance of each VM instruction for each state, this code generation task is simple and fast: the code generator is a deterministic automaton (DFA): the state determines the VM instruction instance to use, and that determines the next state. We can keep this DFA model when we replace some VM instruction instances with sequences of state transitions and VM instruction instances.

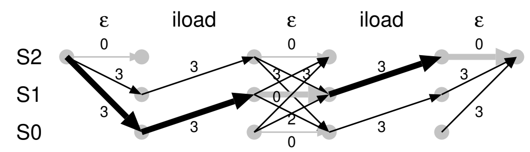

Figure 5. Generating optimal code for `iload iload`, starting in S2, and ending in S2.

However, in general there are several applicable combinations of state transitions and VM instruction instances, turning the automaton into an NFA. Which combination is best depends on later VM instructions.

E.g., consider Fig. 4 again: Starting in S2, we have to generate code for an `iload`; we have to use a transition, either to S1 or to S0. Let us assume that both of these transitions have the same cost, and that both implementations of `iload` also have the same cost. After the `iload` we would get either S1 or S2. If the next instruction is an `iload`, we prefer S1 (otherwise we need to insert another state transition to get to S1); if the next instruction is an `iadd`, we prefer S2.

Trying to generate optimal code for a sequence of VM instructions is a shortest-path problem (see Fig. 5) and can be solved with dynamic programming. If faster code generation is desired, we can also solve the shortest-path problem with a fast Burg-generated automaton [Pro95]: represent states as nonterminals, VM instructions as (unary) operators, and the state transitions as chain rules.[4]

For a simple set of cache states and a pretty complete set of VM instruction instances, the benefit of using the optimal algorithm is small and usually will not amortize the additional time spent in the front-end, but for more complex stack-cache organizations and sets of VM instruction instances with many holes or additional alternatives, the run-time benefit of the optimal algorithm is greater.

However, the main benefit of the optimal approach is that it is easier to implement a shortest-path algorithm (where the only thing you have to specify are the costs) than to generate a good DFA in the more complex scenarios.[5]

4 Implementation

This section describes the implementation of stack caching in Gforth.

[4] Both approaches produce a shortest path, and therefore code of the same quality. Using Burg results in a faster code generator, but the generated automaton is specific to a specific set of VM instructions instances. For our experiments we used dynamic programming, because it allowed us to choose the set of states, VM instruction instances, etc. on interpreter startup.

[5] As a practical example, we accidentally used a stack cache organization with three “registers” on the IA32 architecture, but the third “register” was allocated to memory. Our shortest-path algorithm automatically avoided VM instruction instances that used this third “register” (because this would increase native-code size, our path-length metric).

4.1 Generator

The code for executing VM instructions and various other code dealing with VM instructions is generated from VM instruction specifications by vmgen [EGKP02]. A VM instruction specification looks like this:

```
iadd ( i1 i2 -- i )
i = i1+i2;
```

The first line specifies the name and the stack effect of the VM instruction, and the generator uses it to generate code for accessing the stack and stack pointer updates, among other things. The other lines are plain C code that is inserted in the generated code after the code for fetching values from the stack and before storing the results on the stack.

We added stack caching to vmgen. Since stack accesses and stack pointer updates are implicit in the VM instruction specifications, these specifications did not have to be changed. At the moment the additional specifications needed are:

- A description of the stack-cache states: real-machine registers (actually C variables), and for each state, which registers are used in the state, in which arrangement, and what the stack pointer offset is in that state.
- Specifying that VM instruction instances for various input and output states should be generated. The following line does that for `iadd` starting in S2 and ending in S1:

  ```
  \E S2 S1 state-prim iadd
  ```

  However, this is not normally used. Instead, the following line produces instances for all state combinations useful in a simple stack organization (see Section 3.2):

  ```
  \E prim-states iadd
  ```

In the long term, the plan is to make specifying stack caching for simple cache organizations almost completely automatic, and we are pretty confident that we can achieve this goal.

The code produced for executing `iadd` starting in S2 and ending in S1 looks like this (simplified):

```
I_p17: /* iadd ( i1 i2 -- i ) S2 -- S1  */
{
int i1, i2, i;
i1 = reg1; /* read "registers" */
i2 = reg2;
/* no stack pointer update */
{ /* C code from specification */
i = i1+i2;
}
reg1 = i; /* write "register" */
NEXT; /* dispatch */
}
```

The assignments between the "registers" like `reg1` and the named stack items like `i1` are there because this makes the generator simpler (no replacement in the user-provided C code necessary), avoids restrictions on how the named stack items can be used, and allows having several different types for the named stack items. These assignments are expected to be optimized away by the C compiler.[6]

Overall, the non-dispatch code generated on the PPC for this example looks like this:

```
add     r19,r19,r15
```

4.2 Integration with Gforth

Previous sections often described a variety of possible ways to use stack caching. This section looks at a concrete implementation: stack caching in Gforth, a product-quality implementation of the Forth language (the stack-caching variant has not yet been released).

For each VM instruction Gforth provides a slot for a threaded code address, but it does not provide slots for state transition code. The original reason for this is to support image files that are independent of the VM interpreter used for running them, in particular independent of the organization of the stack cache, if any.

An additional benefit is that the changes for adding stack caching were limited to a relatively small part of the system that deals with low-level threaded-code generation, with only one exception: slightly higher-level code has to report VM-level basic block ends due to control flow joins or procedure boundaries.

However, there are also some drawbacks: if a non-relocatable VM instruction instance is at the start or at the end of a basic block, or adjacent to another non-relocatable VM instruction instance, there is no way to execute a state transition before or after the VM instruction. In this environment using the shortest-path algorithm ensures that we can generate code; a DFA-based code generator could get into the wrong state at the end of a basic block, possibly with no way to get back to S1.

Gforth uses a stack cache organization as in Fig. 2, with more registers and correspondingly more states on some machines. Gforth returns to S1 at every basic block boundary; since calls and returns constitute the majority of basic block boundaries in Forth [Ert95], a more complex way of reconciling the states at control flow joins would provide little benefit, unless we do it interprocedurally. Returning to S1 is suboptimal if three or more registers are available, but is necessitated by some historical baggage.

For all VM instructions, Gforth has instances that start in S1 and end in S1. This ensures that there is always a path for the shortest-path algorithm to find, even if non-relocatable instructions make it impossible to use state transitions. In addition, Gforth has instances of the frequently-occurring VM instructions for the other states, typically at most one version per state, as outlined in Section 3.2.

The shortest-path algorithm in Gforth weighs each edge by the length of the real-machine code in the VM instruction implementation. Overall, this strategy minimizes the length of the generated code, which is usually also helpful for running time.

Traditionally, Gforth loads an image file containing a relocatable variant of threaded code at startup. Later the user can enter Forth source code interactively, or load it from files, and Gforth compiles it into threaded code as soon as it sees it.

Gforth with dynamic superinstructions and stack caching processes the same image file format and the same source code; the reloca-

[6] In this example gcc-2.95 manages to optimize all of them away on the PPC, but generates one `mov` instruction on the 386 architecture.

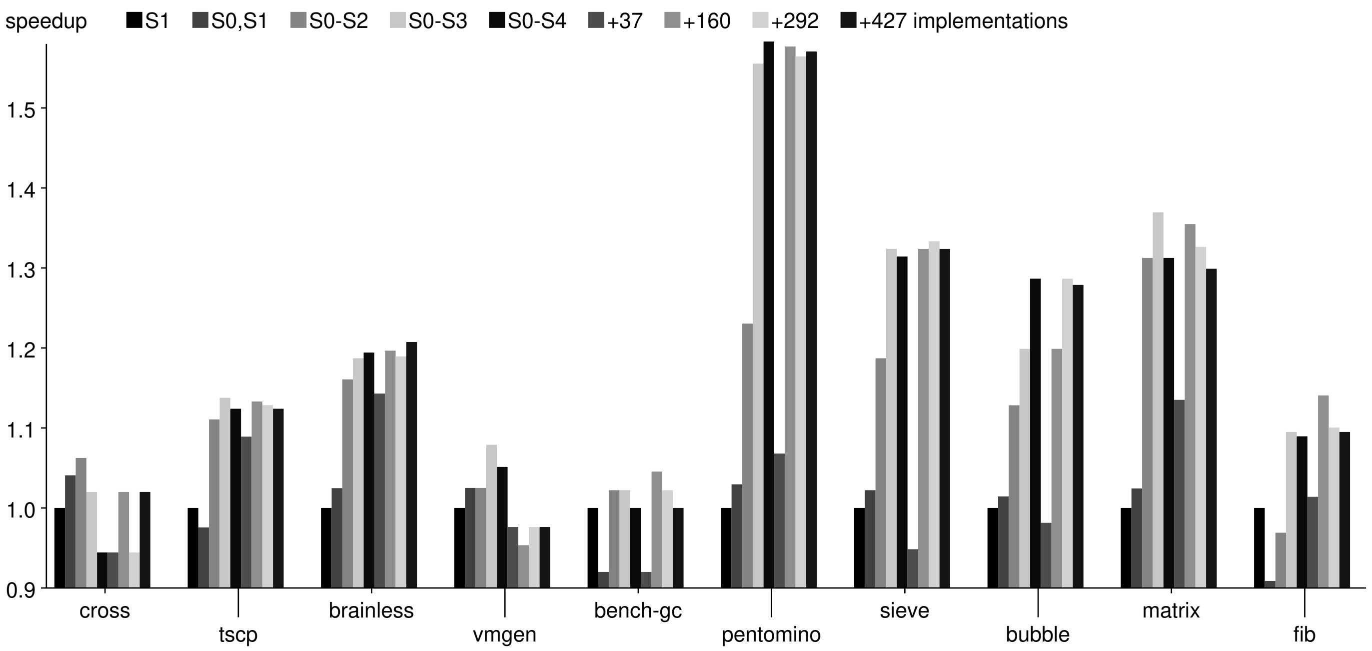

Figure 6. Speedups by using stack caching for various benchmarks

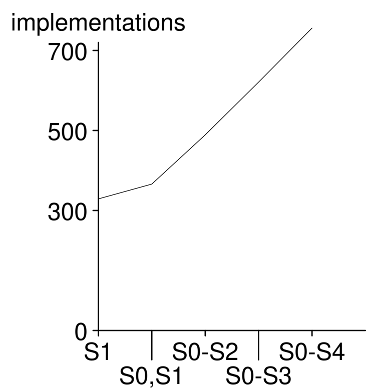

Figure 7. VM instruction instances for stack caches with varying numbers of states

tion part of the image loader and the threaded-code generator of the Forth compiler were modified to produce dynamic superinstructions with stack-caching instead of ordinary threaded code (i.e., these steps occur once for each piece of code). This required relatively few and local modifications in the existing code base. The main complication was that we had to buffer up to a basic block of VM code before we could run the shortest-path algorithm and generate the actual code (both on the threaded-code and the native-code level).

5 Experimental Results

We conducted our experiments on a PowerMac G4 with a 450MHz PPC7400 with 256MB RAM, running under Linux 2.4.25. We used a stack cache organized like in Fig. 2, but with two additional registers (i.e., 4) and two additional states (i.e., 5).

Program	Version	Lines	Description
cross	0.6.9	3793	Forth cross-compiler
tscp	0.4	1625	chess
brainless	0.0.2	3519	chess
vmgen	0.6.9	2641	interpreter generator
bench-gc	1.1	1150	garbage collector
pentomino		516	puzzle solver
sieve		23	prime counting
bubble		74	bubble sort
matrix		55	integer matrix multiply
fib		10	double-recursive function

Figure 8. Benchmark programs used

Gforth has 329 VM instructions. We chose a set of about 140 simple VM instructions for which we generated additional instances for the additional states, resulting in a total of 407 additional implementations (as discussed in Section 3.2, we do not generate an instance of every VM instruction for every state) plus 20 state transition routines.

To simulate using fewer states or fewer VM instruction instances, we provided command-line options that restrict the shortest-path algorithm from using some states or a user-specified set of VM instruction instances. Eliminating states reduces the number of VM instruction instances used as shown in Fig. 7.

The benchmarks we used are shown in Fig. 8. We ran them on Gforth configured with varying amounts of stack caching. Figure 6 shows timing results for these benchmarks. We varied either the number of states (S1...S0-S4) or allowed up to five states, but with a more limited number of VM instruction instances, including transitions (+37...+427).

The number of additional instances in the instance-restricted runs

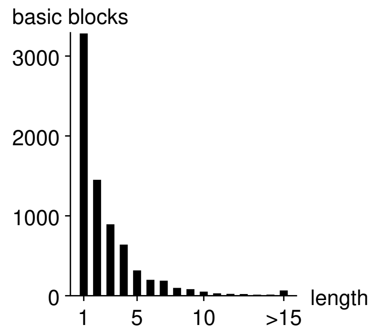

Figure 9. Static basic block lengths for brainless

corresponds to the number of additional instances available in the various state-restricted runs (i.e., the S0,S1 run had 37 additional instances available, the S0-S2 run 160, and the S0-S3 run 292; the S0-S4 run uses the same set of instances as the +427 run). The instances eliminated for the instance-restricted runs are the statically least-frequently used ones for brainless (plus the Gforth library). The elimination is done successively, reevaluating the usage every time an instance is eliminated.

The state-restricted runs are relevant for register-starved architectures: e.g., on the 386 architecture Gforth can use only one register in the threaded-code version or two for the native-code compiler (under development). The instance-restricted runs are relevant for cases where the number of additional instances is limited, e.g., because compiling on small machines is required (gcc-2.95 needed 90MB for compiling Gforth with 427 additional instances), or because there are other consumers of additional VM instruction instances, e.g., specialized VM instructions or static superinstructions.

Our interpretations of the timing results are as follows:

- Cross, vmgen, and bench-gc are short-running programs (about 0.5s), and the speedups from stack caching are barely sufficient to amortize to additional costs incurred by the additional instances and states in our unoptimized implementation of the shortest-path algorithm (0.03s–0.04s for S0-S4 over S1). You also see some random timing variations in these benchmarks (e.g., S0-S4 and +427 should give the same results, but they differ by 0.04s for cross).
- Among the longer-running programs, tscp and brainless are written in a typical Forth style, with a high call frequency resulting in very short basic blocks (see Fig. 9). Fib (a mini-benchmark) also has a high call frequency and short basic blocks. Under these circumstances, static stack caching as implemented in Gforth frequently returns to S1, reducing the benefits of stack caching, and limiting the speedups to factors of 1.15–1.2. We need either interprocedural stack caching or basic-block-extending transformations like inlining to get around this problem.
- Pentomino, sieve, bubble, and matrix have longer basic blocks; sieve, bubble, and matrix are not in typical Forth style (they were translated from other languages), whereas pentomino spends much of its time in run-time-generated code (which does not look like typical Forth code, either). However, while these programs may not be typical Forth programs, the results for them are probably more representative of VMs like the JVM that typically see longer basic blocks. Here we see much higher benefits from stack caching, up to a factor of 1.58 for pentomino.

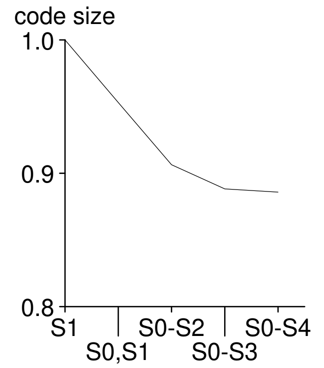

Figure 10. Real-machine code size for varying numbers of states for brainless

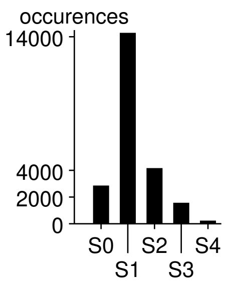

Figure 11. Static state usage in brainless

- The S0,S1 organization gives very little and sometimes even negative speedup compared to S1 single-state caching. However, it reduces the real-machine code size very effectively (see Fig. 10). We believe that most of the speedup of higher-state stack caching on our benchmark machine comes from not having to wait for loads to complete before operating on them. The S0,S1 organization does not have this benefit over the S1 base organization we used, so we do see very little speedup; the reduction in code size and executed instructions apparently has little effect.
- S0-S2 gives good results most of the time, but sometimes S0-S3 is significantly better. S0-S4 does not perform better than S0-S3, because S4 is rarely used (see Fig. 11).
- Using just 37 additional implementations often produces little benefit, probably for the same reason that makes S0,S1 relatively slow.
- Using 160 additional implementations usually produces the same benefit as using more implementations, sometimes outperforming S0-S2 (with the same number of implementations) significantly.

Looking at the code size benefit of stack caching (Fig. 10), we see an overall reduction by a factor 1.13 for brainless.

6 Related work

Stack caching was first published by DeBaere and Van Campenhout [DV90], who presented a small example of dynamic stack caching.

Ertl [Ert95] discussed stack caching in more detail, including various stack cache organizations, static and dynamic stack caching, and presented results in numbers of eliminated loads, stores, and stack pointer updates, but produced no full implementation.

Sun's Hot Spot JVM system performs dynamic stack caching in its interpreter part [Gri01]: It caches up to one stack item in registers; for each of the four types (int, long, float, double), it has a separate state that represents the presence of one stack item of this type in registers (different registers are used for some of these types). It is not necessary to implement instances of all instructions for all states, because the type rules of the JVM disallow many state/instruction combinations.

Ogata et al. [OKN02] implemented dynamic stack caching with up to two registers, but eventually dropped it because the speedup from that on their Power3 machine was not large enough (1%–4% over single-state stack caching) to justify the complexity.

The differences between the present paper and these papers is that we present an implementation of static stack caching, we evaluate it using substantial programs as benchmarks, and we also discuss various other issues in detail, in particular reducing the number of VM instruction instances, and using a shortest-path algorithm for code generation.

Peng et al. [PWL04] introduce a technique for saving real-machine code space in static stack caching (with an unconventional stack cache organization) by arranging the code for the VM instruction instances such that they share one piece of code, with different entry points for the different instances. The difference between this paper and our paper is that we combine static stack caching with dynamic superinstructions, that we use different and more stack cache organizations (designed for execution speed, not code sharing), and that we use a faster single-state baseline (S1) in our empirical work.

Dynamic superinstructions were first mentioned by Rossi and Sivalingam (as *memcpy method*) [RS96]. Piumarta and Riccardi [PR98] expanded on that and implemented this idea in a full-scale interpreter. We [EG03] showed that dynamic superinstructions are not just simpler, but also faster (due to better branch prediction), if you do not eliminate duplicate copies of the same superinstructions.

In this paper, we use dynamic superinstructions mainly as the environment for implementing stack caching; the presence of dynamic superinstructions influences some design decisions, in particular it makes state transitions much cheaper, allowing to reduce the number of VM instruction implementations.

7 Conclusion

Dynamic superinstructions work well with static stack caching. They eliminate the dispatch overhead of state transitions, allowing to reduce the number of VM instruction instances without increasing the execution cost. Static stack caching can use a shortest-path algorithm to generate optimal the optimal sequence of VM instruction instances and state transitions for a given sequence of VM instructions.

We implemented these ideas in the interpreter generator vmgen, automating the work of producing the various VM instruction instances, and in the interpreter Gforth. As a result, we see speedups of 15%–20% for programs written in typical Forth style with very small basic blocks, and 30%–58% for benchmarks with larger basic blocks, which may be more representative of the results you may see for other languages. Using four states with three registers and 160 additional VM instruction implementations is enough to achieve nearly all of these speedups.

Acknowledgements

The anonymous reviewers provided helpful comments.

8 References

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

[DBC+03] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The case for virtual register machines. In *Interpreters, Virtual Machines and Emulators (IVME '03)*, pages 41–49, 2003.

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

[EG03] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN '03 Conference on Programming Language Design and Implementation*, 2003.

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

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

[Gri01] Robert Griesemer. Interpreter generation and implementation utilizing interpreter states and register caching. Patent 6192516 B1, US, 2001.

[OKN02] Kazunori Ogata, Hideaki Komatsu, and Toshio Nakatani. Bytecode fetch optimization for a Java interpreter. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 58–67, 2002.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[Pro95] Todd A. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, May 1995.

[PWL04] Jinzhan Peng, Gansha Wu, and Guei-Yuan Lueh. Code sharing among states for stack-caching interpreter. In *IVME '04 Proceedings*, 2004.

[RS96] Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Faculty of Information Technology, Helsinki University of Technology, May 1996.

[SC99] Vítor Santos Costa. Optimising bytecode emulation for Prolog. In *LNCS 1702, Proceedings of PPDP'99*, pages 261–267. Springer-Verlag, September 1999.