Stack Caching in Forth

M. Anton Ertl[*]
TU Wien

David Gregg
University of Dublin, Trinity College

Abstract

Stack caching speeds Forth up by keeping stack items in registers, reducing the number of memory accesses for stack items. This paper describes our work on extending Gforth's stack caching implementation to support more than one register in the canonical state, and presents timing results for the resulting Forth system. For single-representation stack caches, keeping just one stack item in registers is usually best, and provides speedups up to a factor of 2.84 over the straight-forward stack representation. For stack caches with multiple stack representations, using the one-register representation as canonical representation is usually optimal, resulting in an overall speedup of up to a factor of 3.80 (and up to a factor of 1.53 over single-representation stack caching).

1 Introduction

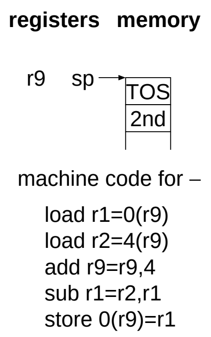
In threaded-code interpreters for Forth, and especially in simple inline-expanding native-code compilers a significant part of the run-time is consumed by loading stack items from and storing them to memory, and by stack pointer updates.

A frequent technique for reducing that overhead is to keep the top-of-stack in a register. Stack caching [Ert95] is a generalization of this technique. In the past we have presented data based on simulations [Ert95], and timing data with restricted forms of stack caching: Gforth was only able to perform single-state stack caching with one register, and static stack caching with the canonical state containing 0 or 1 registers [EG04].

In this paper, we describe how we lifted these restrictions (Section 3), and present empirical results, including timing results for several different machines (Section 4).

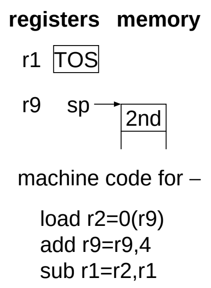
2 Background

This section gives an overview of stack caching [Ert95, EG04].



```
load r1=0(r9)
load r2=4(r9)
add r9=r9,4
sub r1=r2,r1
store 0(r9)=r1
```

Figure 1: A straight-forward representation of the stack



```
load r2=0(r9)
add r9=r9,4
sub r1=r2,r1
```

Figure 2: Keeping the top-of-stack in a register

2.1 Stack representation

A straight-forward representation of the stack is to keep all stack items in memory, and have a stack pointer that points to the top-of-stack (Fig. 1). This requires a memory access for every stack accessed stack item.

A frequently used improvement over the straight-forward representation is to keep the top-of-stack in a register (Fig. 2). This makes the frequent accesses to the top-of-stack substantially cheaper.

2.2 Using several registers

One might consider keeping more stack items in registers all the time. However, this does not necessarily lead to an improvement in running time, because with many stack items in registers, changing the stack depth often requires additional moves between registers (Fig. 3). Whether more stack items provide a speedup, depends on the mix of primitives,

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

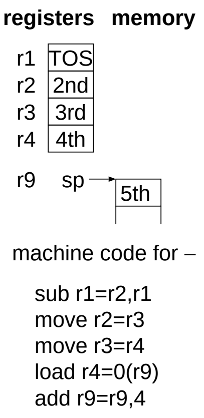


Figure 3: Keeping the four top stack items in registers

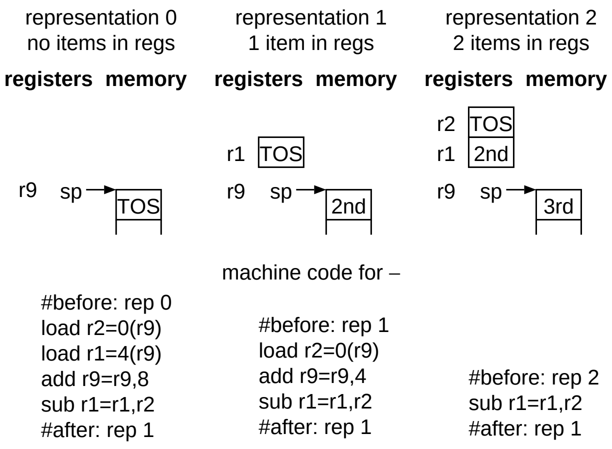


Figure 4: A stack cache with multiple stack representations

and on the characteristics of the machine executing the code.

In the past we have presented only simulation results for stack caches with more than one register. In this work we present timing results from real machines.

2.3 Multiple stack representations

To avoid the cost of the register moves (and other costs) when changing the stack depth, one could change the stack representation during the execution of a primitive (Fig. 4); note how cheap `-` becomes when it starts in representation 2 and is allowed to finish in representation 1. Of course, then the next primitive executed has to be in a version that starts in representation 1 (or we have to insert additional code that switches between representations). So, in order to make profitable use of this, we need different implementations of at least the common primitives, for different stack representations.

2.4 Static stack caching

How do we get the right version of the primitive to execute? There are at least two ways, but the more promising one is static stack caching: The compiler keeps track of the stack representation, and for each primitive it has to compile, it compilers an appropriate version of the primitive.

This approach requires that the stack representation is the same when two control flow paths join. Moreover, for simplicity in the compiler it is best if the stack representation is the same at all points where control flow can happen (in compiler terminology, at all *basic block* boundaries); this representation is the *canonical stack representation.*

In earlier work, we only had simulation results for static stack caching [Ert95], or timing results where the canonical stack representation could have at most one register [EG04]. In the present work, we present timing results for stack caches with other canonical states.

3 Implementation

3.1 Interpreter generator

The code for Gforth's primitives is written in a mixture of Forth and C. E.g., here is the code for the primitive `+`:

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

An interpreter generator [EGKP02] translates this code into (GNU) C code, and gcc then translates it into an executable interpreter.

One important aspect of the interpreter generatore is that it generates all the stack access code for a primitive from the specification of the stack effect in the first line of the primitive's specification.

To implement stack caching, we generalized the access-generating code to deal with arbitrary stack representations, including different representations before and after the primitive. We also added ways to specify stack representations, and to determine which versions of a primitive are generated.

One problem in this context were primitives that access a stack pointer explicitly in their C code, either because they have to manipulate it (e.g. `sp!`), or because they do something beyond the descriptive powers of the stack effect specification in the interpreter generator (e.g., `?dup`). The primitives affected in Gforth are: `sp@ sp! fp@ fp! ?dup ?dup-?branch ?dup-0=-?branch pick >float fpick` and some C call interface primitives.

In our first foray into multi-state stack caching [EG04], we just left these primitives alone, so that they would just keep working with 0 or 1 stack items in registers. However, this restricted the canonical stack representations we could use to just those with 0 or 1 stack items in registers.

In the present work, we eliminated this restriction: You can now put the string `...` (possibly prefixed by a stack prefix) into the stack effect description of a primitive; this causes the generator to flush all the cached stack items to memory and let the stack pointer(s) point to the top-of-stack, thus presenting the C code with the straightforward stack representation (Fig. 1); after the C code, stack items are loaded into registers and the stack pointer is adjusted as is necessary for the representation after the primitive. Here is an example:

```
pick ( S:... u -- S:... w ) core-ext
w = sp[u];
```

Here the `S:...` indicates that the data stack has to be flushed before and reloaded after the primitive. An additional advantage of this approach is that these primitives became much easier to understand than they used to be; before this extension, one had to consider the kind of code that the generator would produce, often with conditional compilation for dealing with the differences between using 0 or 1 register.

3.2 Code generator

When Gforth compiles Forth code (or loads the system image), it has to select which versions of the primitives (out of several with different input and output stack representation) should be used. This selection is performed by C code that hooks into the Forth compiler via `compile,` and is also called from the loader. This code generator uses a shortest-path algorithm for selecting the optimal sequence of primitive versions (optimality criterion: minimum sum of the native-code sizes of the primitive versions). This code generation process is described in more detail in our earlier work [EG04].

3.3 Effects on Forth code

To work correctly with stack caching, the colon definitions must not access stack items in memory (with `sp@` and memory operations). Fortunately, there was only one colon definition in the Gforth system that did this: `roll`. This definition was changed into one that does not use `sp@` and does not use memory operations to access stack items.

In addition to that, there were some very small changes to make the static stack caching code generator (written in C) aware of control flow joins (`then`, `begin`).

These were the only changes that were needed in the Forth code of the Gforth system, so the changes for static stack caching were fairly local.

3.4 GCC issues

Stack caching introduces additional versions of the primitives. The versions of Gforth we used for the present work contain around 1200 primitives and their versions: 355 basic primitives (starting and ending in the canonical state), 795–848 versions of popular primitives for other transitions between stack representations, and 13 superinstructions (deactivated in our experiments).

In older versions of GCC and with our old way of coding NEXT in the primitives, having so many primitives and their versions resulted in a huge memory consumption (several hundred MB) and long compile times (on the order of a half-hour).

With more recent GCC versions, this problem was not present, but they generated code that disabled dynamic superinstructions, a very profitable optimization in Gforth that is also essential for our implementation of static stack caching.

We worked around both of these problems by changing the way we code NEXT. Instead of appending the NEXT sequence including an indirect `goto` (`goto *`) to each primitive, we just have one indirect goto (very early) in the whole function. At the end of each NEXT, we append a direct goto to this indirect goto:

```
engine(...)
{
   ...
  before_goto:
   goto *real_ca; /* indirect goto */
  after_goto:
   ...
  I_plus:
   ... /* all of + except NEXT */
   ip++; /* maintain ip for accessing
            immediate arguments */
  K_plus:
   real_ca = ip[-1]; /* NEXT, part 2 */
  J_plus:
   goto before_goto;
  ... /* other primitives */
}
```

For dynamic superinstructions, when we want to generate the code for a `+` without a NEXT, we copy the code between `I_plus` and `K_plus` to the native-code area of the current definition. But if we want to include the NEXT (normally that only happens for branching primitives), we copy the code between `I_plus` and `J_plus`, and append the code between `before_goto` and `after_goto`; this avoids the problems with the non-relocatability of the `goto before_goto`.

The benefit of this workaround in our context is that even older gccs compile `gforth-fast` with the 1200 primitive versions in around a minute (on a

1066MHz PPC7447A), using about 50MB of RAM. With newer gcc versions we get engines where dynamic superinstructions work.

The downside of this workaround is that, if dynamic superinstructions are disabled for some reason, the the Forth system runs significantly slower than the old version of Gforth would run when compiled with an older version of gcc: The additional direct branch per primitive costs time; and on CPUs with branch target buffers (e.g., various Pentiums and Athlons), the shared indirect branch has significantly worse branch prediction than the separate indirect branches had. However, ideally dynamic superinstructions are enabled in all situations where performance is important, so this disadvantage should not be a problem.

4 Results

4.1 Hardware

The main component that determines the performance in our benchmarks is the CPU. We used three different hardware platforms with different CPUs: a 450 MHz PPC7400 (PowerMac G4), a 1066MHz PPC7447A (iBook G4), and a 2000MHz PPC970 (PowerMac G5). The PPC7400 is a shallowly pipelined CPU (4 stages in the integer pipeline) that can issue up to two instructions per cycle; the PPC7447A is a deeper (7 stages) and wider (triple-issue) CPU; and the PPC970 is very deep (16 stages) and very wide (five-issue).

So we can expect to see some performance differences from these CPUs, even though they have the same architecture. We use the PPC architecture for our experiments, because gcc is able to allocate many registers for the stack cache on this architecture, unlike on other architectures we have tried (Alpha, MIPS, AMD64, ARM); we believe that this is caused by the much higher number of callee-saved registers in the PPC calling convention compared to other calling conventions.

All of these machines were running Linux, and we benchmarked the same executable programs on all of them.

4.2 Forth systems

We built nine Gforth engines, all of them with 8 registers usable for stack caches. The engines differ in the canonical stack representation they support, one for each number of registers (0–8). The other stack representations can be controlled using a command-line parameter. E.g., we ran the engine built for the canonical state with three registers with just one stack state to get results for single-representation stack caching with three registers. We also ran it restricted to the representations with 0–3 registers with the three-register representation being canonical; similarly for 0–4 registers up to 0–8 registers. In this way the 9 basic engines were used for evaluating 53 stack caching organizations.

Even though we built the engines with a few static superinstructions, we disabled them in benchmarking, because the combination of static stack caching and static superinstruction is not supported yet, so the static superinstructions just work in the canonical state, and enabling them might suppress some of the effects of stack caching (to a greater extent than an proper combination of stack caching and static superinstructions would).

The engines were built with gcc-4.0.1 (Debian 4.0.1-2).[1]

Program	Vers.	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
CD16sim	1.1	937	CPU emulator
brew	t_38	31401	evolutionary playground
pentomino		516	puzzle solver
sieve		23	prime counting
bubble		74	bubble sort
matrix		55	integer matrix multiply
fib		10	double-recursive function

Figure 5: Benchmark programs used

4.3 Benchmarks

Figure 5 shows the benchmarks we used for our experiments. In addition to timing results, we also present instruction, load, and store counts; they were collected using the performance monitoring counters of the PPC7447A and the `perfex` utility of the perfctr patch for Linux. We use the same executables on all machines, so the number of executed instructions, loads, and stores are the same on all of them. We ran each benchmark three times for each configuration, and present the median of the three runs.

4.4 Run-time and instructions

Figure 6 shows the number of instructions executed by the benchmark *Brainless*. The line labeled `n`

[1]We suspected that auto-increment load and store instructions combined with the selection of which stack item the stack pointer points to might influence the results, so we also performed experiments with compiling with the `-mno-update` flag, which suppresses generating code that uses auto-increments. However, the results were essentially the same either way, so our suspicion was disproved. In this paper, we report the results without `-mno-update`.

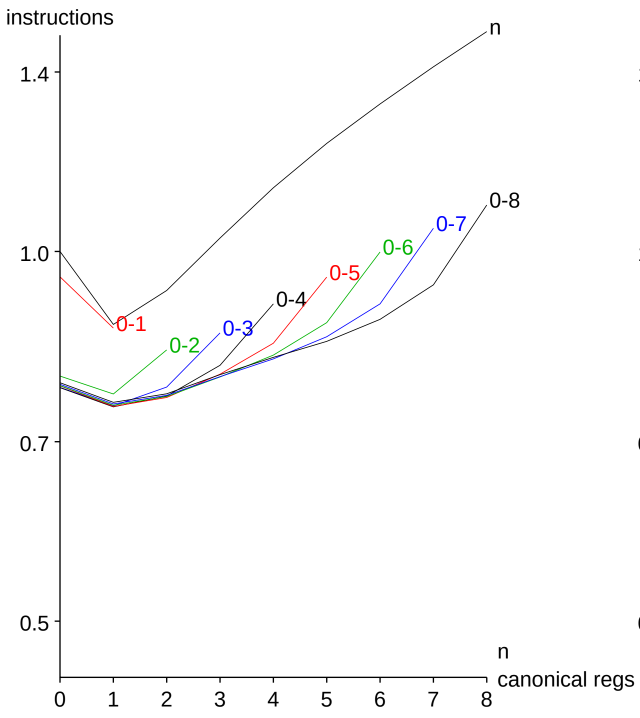


Figure 6: Instructions executed by Brainless

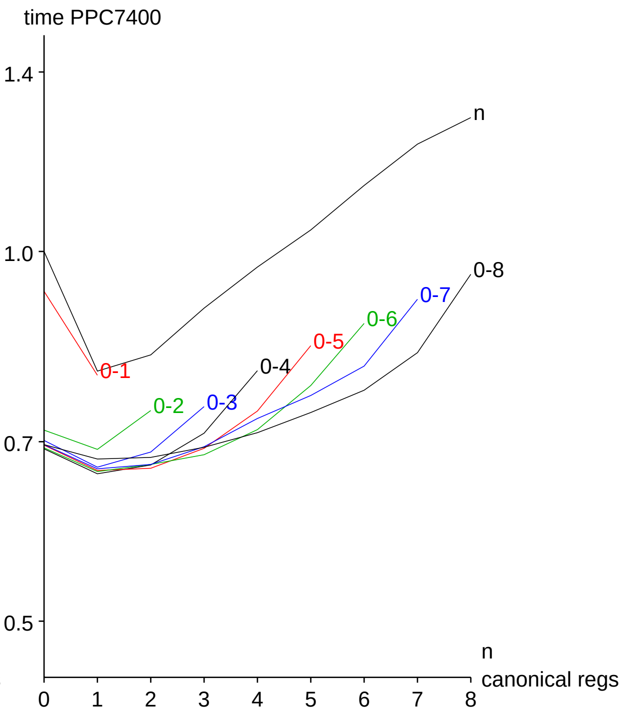


Figure 8: Brainless run-time on PPC7400

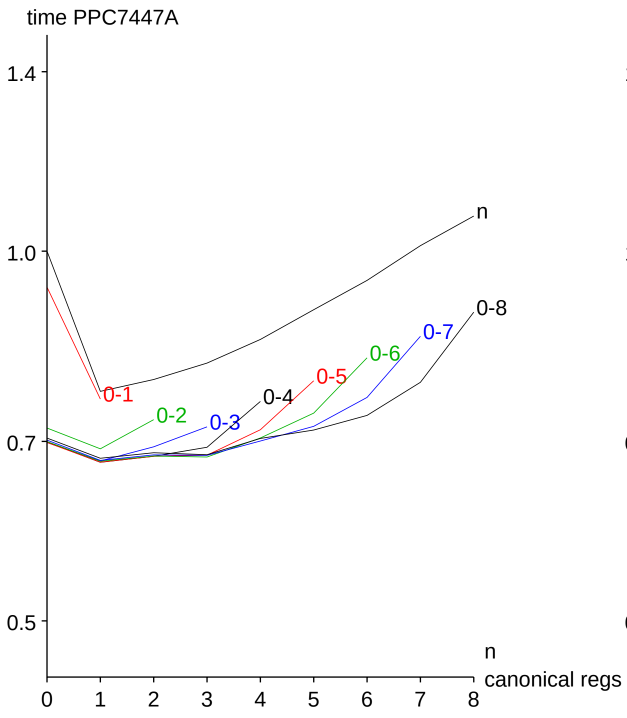


Figure 7: Brainless run-time on PPC7447A

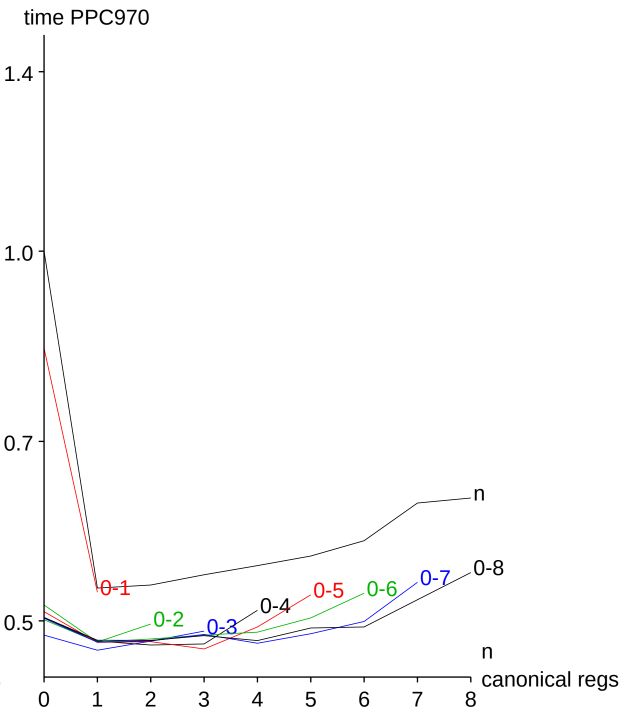


Figure 9: Brainless run-time on PPC970

represents the stack caches with a single stack representation; that stack representation is indicated by the position on the x-axis. The lines labeled 0–x represent stack caches using stack representations with 0 to x registers; the canonical representation is indicated by the position on the x-axis.

Figure 7, 8 and 9 show timing results for Brainless on different CPUs.

Figure 12, 13, 14 and 15 show instruction counts and timing results for all benchmarks; two single-representation results are shown per benchmark: for keeping one stack item in a register all the time, and the best single-representation scheme for the benchmark (this may be different from the best scheme for other benchmarks). Similarly, for the multiple-state schemes the scheme with up to three registers (0-3) with the canonical representation keeping one stack item in a register is shown, and the best multi-representation scheme for the benchmark.

Which canonical representation?

For the multiple-representation stack caches, once the number of registers available exceeds those in the canonical representation by two or more, all caches with the same canonical representation perform about the same. The number of instructions executed is smallest for the canonical stack representation with one register (except for some of the smaller benchmarks). Similarly, for the single-representation stack caches, the one with one register executes the least instructions.

The PPC7400 timings behave quite similar to the instruction counts, although the timing reduction is somewhat higher than the instruction reduction; on the PPC7447A and especially the PPC970 the times for canonical representations with more than one registers rise much more slowly (and sometimes not at all).

Nevertheless, even on those CPUs using the one-register representation as canonical representation or, for single-representation stack caches, as the representation is optimal for many benchmarks, and close to optimal on the others.

How many registers?

With the canonical representation set to using one register, how many registers should be used for a multiple-representation stack cache? More than three registers does not help much (see Section 5 for an explanation); so if three registers are available, they should be used. Two registers are almost as good, but with just one register, the speedup over the one-register single-representation stack cache is tiny.

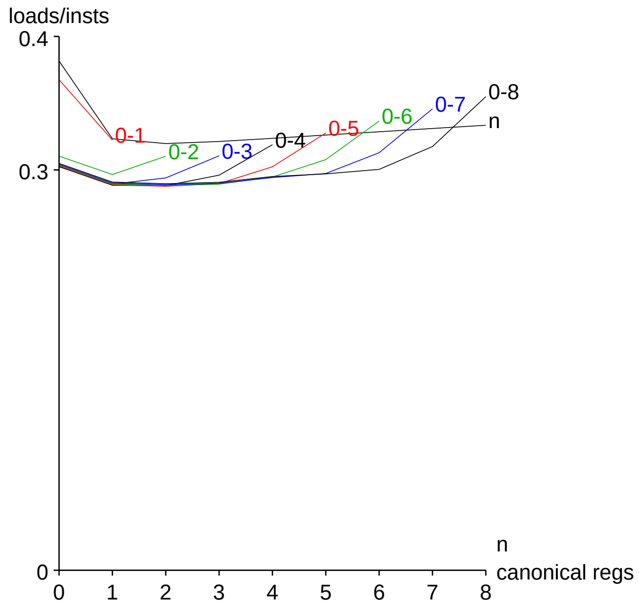


Figure 10: Load instructions executed dynamically by Brainless

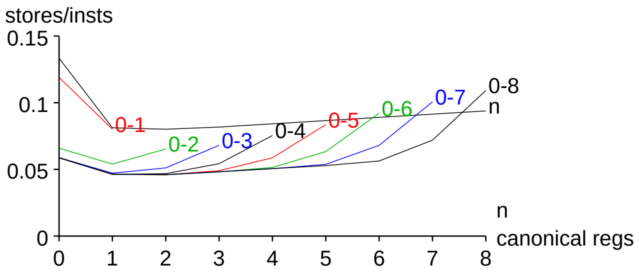


Figure 11: Store instructions executed dynamically by Brainless

Are multiple representations worthwhile?

In the setup we evaluated, the 0-3 stack cache with the one-register canonical representation provides up to a factor of 1.53 speedup (Pentomino on the PPC7400) over the single-representation stack cache with one register. If enough registers are available (at least two), the speedup may well be worth the implementation cost.

Benefit of single representation

While the case for multiple stack representations depends on the circumstances, the case for keeping one stack item in registers all the time is pretty clear. For a tiny increase in implementation complexity we get a significant increase in performance, in particular on the PPC970. In earlier work [EGKP02] we have also tested this on other CPUs; the results were not as spectacular as for the PPCs, but still worthwhile.

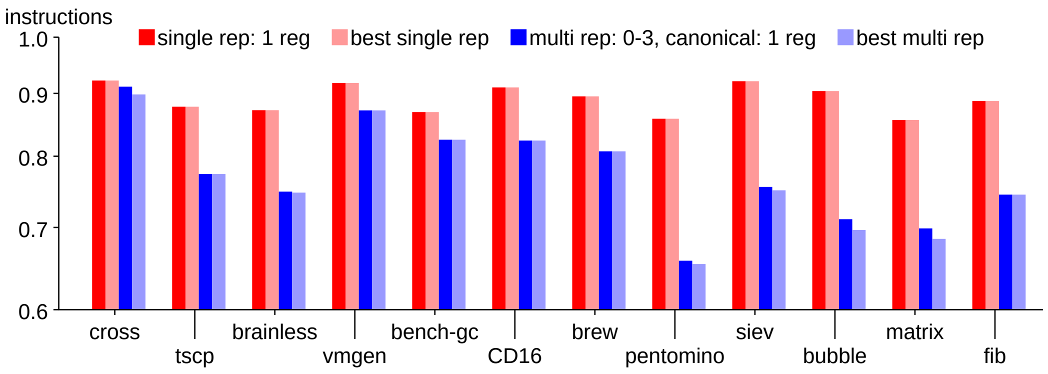


Figure 12: Instructions executed dynamically relative to the straight-forward stack representation

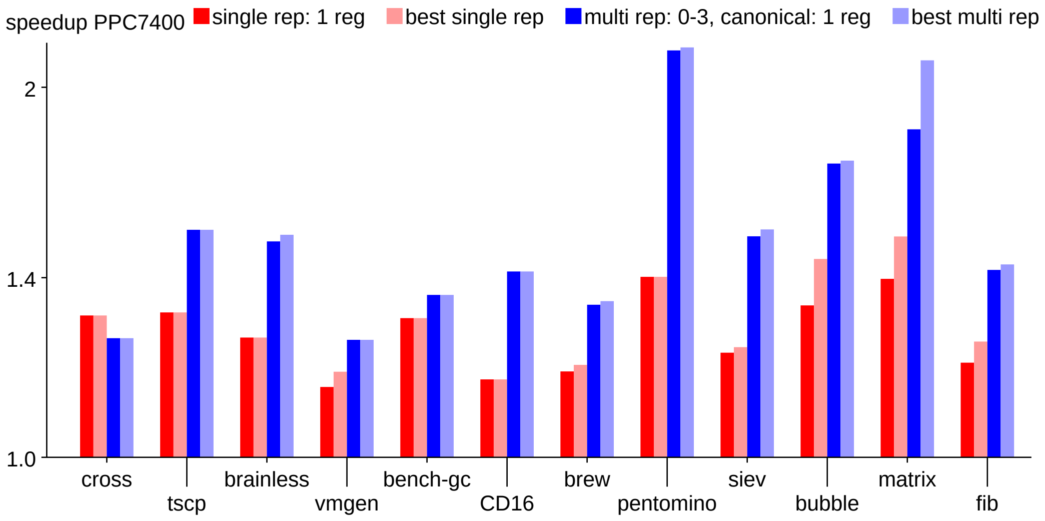


Figure 13: Speedup on the PPC7400 over the straight-forward stack representation

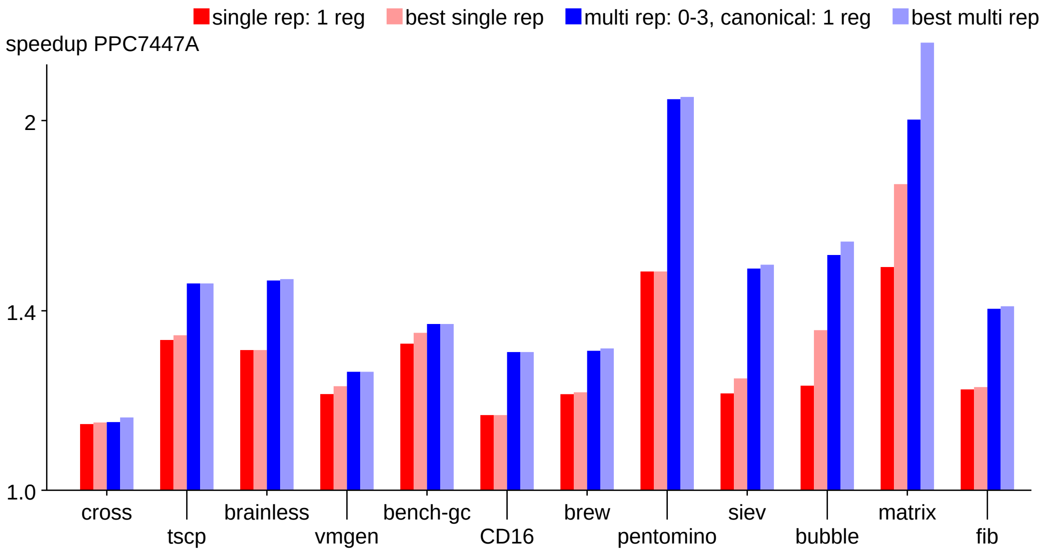


Figure 14: Speedup on the PPC7447A over the straight-forward stack representation

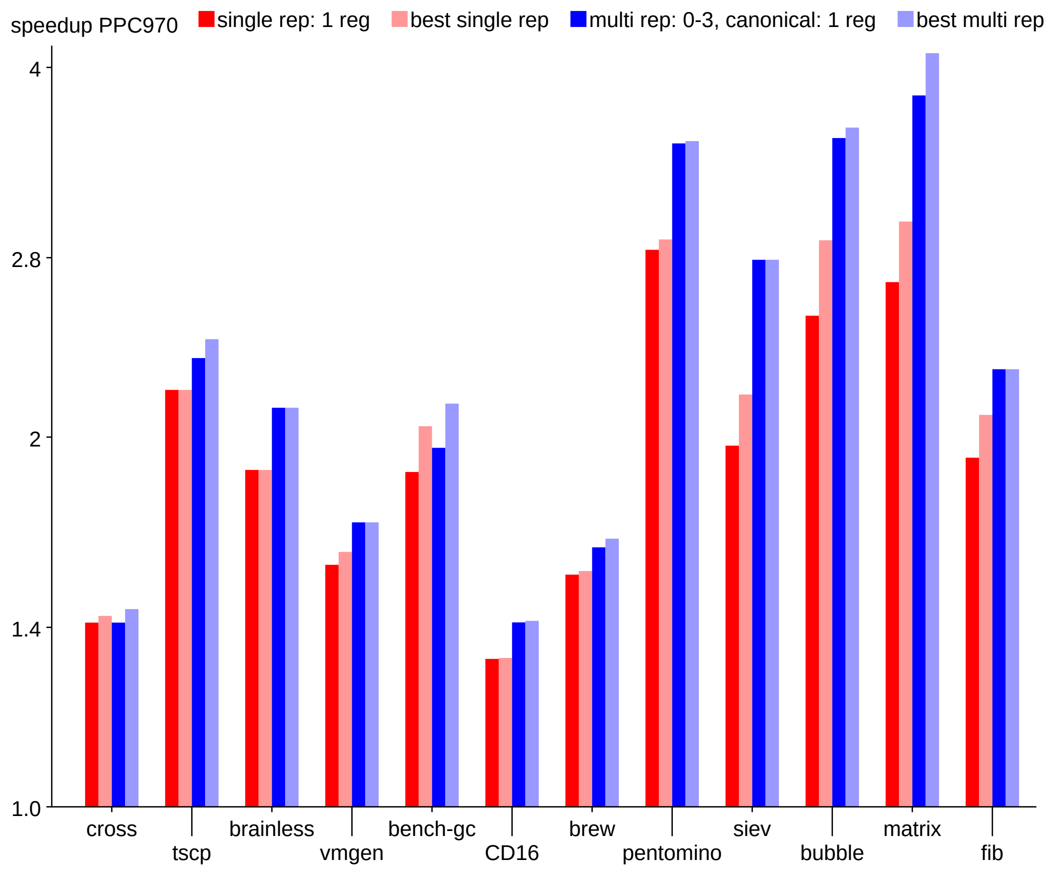


Figure 15: Speedup on the PPC970 over the straight-forward stack representation

Loads and Stores

Figure 10 and Fig. 11 shows the number of executed loads and stores as proportion of the number of executed instructions.

Stack caching reduces both loads and stores by about the same number. However, there is a big baseline of loads that do not perform stack accesses, which is the reason for the difference in the way the pictures look.

One big contributor to this baseline is that each primitive still loads the address of the next one. This is mostly redundant in the context of dynamic superinstructions and could be optimized away.

The significance in the number of loads and stores is that some CPUs have particular performance issues related to these instructions. In particular, there are a number of CPUs that are store-limited, because their writes go off-chip (no on-chip caches, or only write-through on-chip caches); CPUs of this class are the 486DX2 and some 486DX4s, the MicroSPARC II, the 21064 and the 21164PC; for newer high-performance CPUs this is a problem of the past, but it might show up in embedded systems (and sometimes as a bug workaround elsewhere). For store-limited CPUs the speedup can be directly proportional to the reduction in stores.

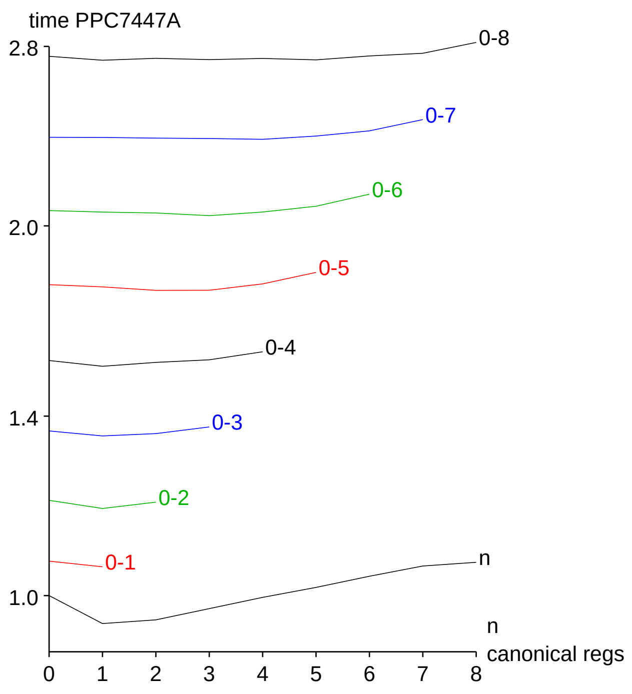


Figure 16: Gforth startup time

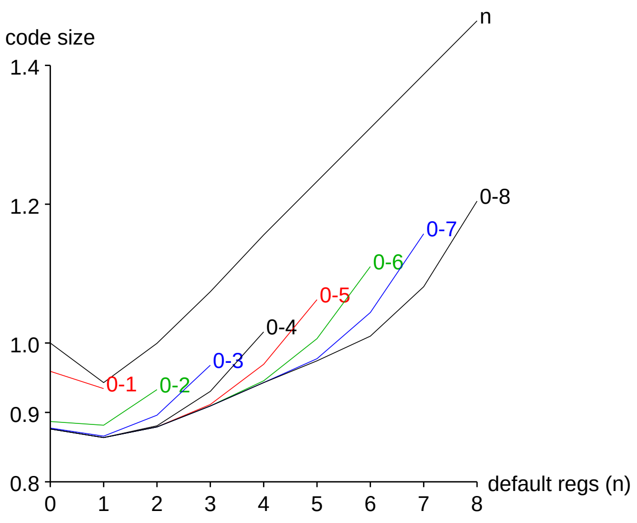


Figure 17: Code size of the dynamically generated native code for the Gforth image

4.5 Compile time

The time taken by the shortest-path algorithm used in the code generator (Section 3.2) takes time linear with the number of stack representations. This affects the startup time of Gforth (where the code generator is applied to the code of the image file), and the compilation speed. Figure 16 shows the resulting changes in the startup time. Note that even with 9 representations, the startup time of Gforth on the 1066MHz PPC7447A is still only 0.05s, so in most applications this is not a serious problem; however, it is visible in the results of short-running benchmarks.

It is possible to have a faster code generator that uses a two-pass automaton and has performance independent of the number of stack representations, but we have not implemented that (yet).

4.6 Code size

The code size is also affected by stack caching (Fig. 17). With a single stack representation with one register, the code is 0.94 times as large as without stack caching. With multiple representations the code size can be reduced to 0.86 times the size without stack caching.

However, the additional primitive versions necessary to make multiple representations effective also should be added to the code size; for the engine with 0–8 registers, with one register for the canonical representation, the additional code size is 26068 bytes for the primitives alone; the additional code size for 0–3 registers would be significantly smaller, probably around 10KB. For the Gforth image alone going from always-1 to 0–3 registers saves 24024 bytes, so multiple-representation stack caching can pay for itself already before compiling any additional code.

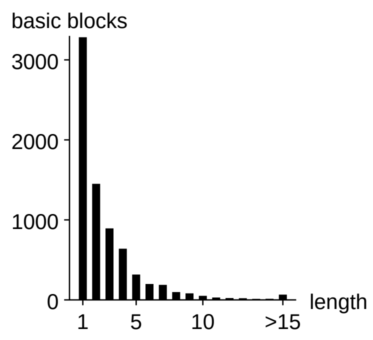


Figure 18: Number of primitives per basic block (static) for Brainless

On the other hand, at least Gforth needs another copy of the additional primitives (for determining relocatability), plus embedded padding, plus some tables describing the additional primitives. And for a smaller image, the savings would be smaller. So multi-representation stack caching does not necessarily reduce the code size.

Moreover, if code size is at a premium, the user would not use dynamic superinstructions, and there would be no code size savings from multiple representations, only the cost of the additional primitives.

5 Further work

The improvement of multiple-representation stack caching over single-representation stack caching is a little disappointing. One reason for this could be that the basic blocks in Forth code are very short, forcing a return to the canonical representation very often (Fig. 18). In particular, for the large number (45% for Brainless) of basic blocks with length one there is no difference between a multiple-representation stack cache and a single-representation stack cache. So, given this basic block length distribution, it is not very surprising that there is not that much performance difference between single and multiple representations.

So if we apply optimizations that make the basic blocks longer, we might see quite different results than those in this paper. For Forth the most promising of these optimizations is inlining [GE04]. We will investigate the effect of inlining in the future.

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.

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 work is that we combine static stack caching with dynamic superinstructions and that we use different and more stack cache organizations (designed for execution speed, not code sharing).

In our earlier work [EG04], we already combined static stack caching with dynamic superinstructions. In this work we expand on that work by implementing stack caching with arbitrary canonical representations, and evaluating the resulting stack cache organizations. We also discuss issues related to high-level Forth code and some issues we had with gcc and how we solved them; also, in the present paper we only give an overview over the code generation topics that were discussed in depth in our earlier papers.

7 Conclusion

For single-representation stack caching, keeping one stack item (the top-of-stack) in a register is usually optimal; the resulting speedup (over using the straight-forward stack representation) depends on the benchmark and the CPU, and can reach up to a factor of 2.84 (pentomino on PPC970); however, on most other CPUs the speedups are significantly smaller.

For multiple-representation stack caching, using a canonical state with one register is often optimal; with that fixed, using more than three registers for the stack cache provides little benefit. This stack cache organization provides speedups of up to a factor 3.80 (matrix on PPC970), but again the results on other CPUs and other benchmarks are often considerably less. The speedup of using this stack caching scheme over single-stack stack caching can reach up to a factor of 1.53 (pentomino on PPC7400). Optimizations that make basic blocks longer (e.g., inlining) might change these results.

References

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

[EG04] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *IVME '04 Proceedings*, pages 7–14, 2004.

[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.

[GE04] David Gregg and M. Anton Ertl. Inlining in Gforth: Early experiences. In *EuroForth 2004 Conference Proceedings*, 2004.

[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.

[PWL04] Jinzhan Peng, Gansha Wu, and Guei-Yuan Lueh. Code sharing among states for stack-caching interpreter. In *IVME '04 Proceedings*, pages 15–22, 2004.