Threaded Code Variations and Optimizations

M. Anton Ertl[*]
TU Wien

Abstract

Forth has been traditionally implemented as indirect threaded code, where the code for non-primitives is the code-field address of the word. To get the maximum benefit from combining sequences of primitives into superinstructions, the code produced for a non-primitive should be a primitive followed by a parameter (e.g., `lit` *addr* for variables). This paper takes a look at the steps from a traditional threaded-code implementation to superinstructions, and at the size and speed effects of the various steps. The use of superinstructions gives speedups of up to a factor of 2 on large benchmarks on processors with branch target buffers, but requires more space for the primitives and the optimization tables, and also a little more space for the threaded code.

1 Introduction

Traditionally, Forth has been implemented using an interpreter for indirect threaded code. However, over time programs have tended to depend less on specific features of this implementation technique, and an increasing number of Forth systems have used other implementation techniques, in particular native code compilation.

One of the goals of the Gforth project is to provide competetive performance, another goal is portability to a wide range of machines. To meet the portability goal, we decided to stay with a threaded-code engine compiled with GCC [Ert93]; to regain ground lost on the efficiency front, we decided to combine sequences of primitives into superinstructions. This technique has been proposed by Schütz [Sch92] and implemented by Wil Baden in this4th [Bad95] and by Marcel Hendrix in a version of eforth. It is related to the concepts of supercombinators [Hug82] and superoperators [Pro95].

Non-primitives in traditional indirect threaded code cannot be combined into superinstructions, but it is possible to compile them into using primitives that can be combined, and then into using direct threading instead of indirect threading. This paper describes the various steps from an indirect threaded implementation to an implementation using superinstructions (Section 2), and evaluates the effect of these steps on run-time and code size (Section 4); these steps also affect cache consistency issues on the 386 architecture, which can have a large influence on performance (Section 3).

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

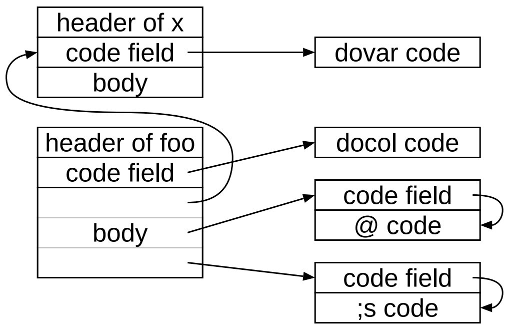

Figure 1: Traditional indirect threaded code

2 Threaded code variations

We will use the following code as a running example:

```
variable x

: foo x @ ;
```

2.1 Traditional indirect threaded code

In indirect threaded code (Fig. 1) the code of a colon definition consists of a sequence of the code field addresses (CFAs) of the words contained in the colon definition. Such a CFA points to a code field that contains the address of the machine code that performs the function of the word.

The reason for the indirection through the code field is to support non-primitives, like the variable `x` in our example: The dovar routine can compute the body address by adding the code field size to the CFA.

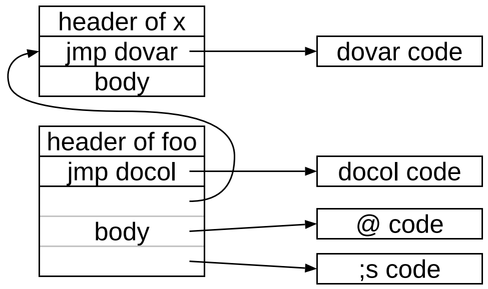

Figure 2: Direct threaded code, traditional variant

2.2 Traditional-style direct threaded code

For primitives the indirection is only needed because we do not know in advance whether the next word is a primitive or not. So, in order to avoid the overhead of the indirection for primitives, some Forth implementors have implemented a variant of this scheme using direct threaded code (see Fig. 2). For primitives, the threaded code for a word points directly to the machine code, and the execution token points there, too.

For non-primitives, the threaded code cannot point directly to the machine-code routine (the *doer*), because the doer would not know how to find the body of the word. So the threaded code points to a (variant of the) code field, and that field contains a machine-code jump to the doer.

This change replaces a load in every word by a jump in every non-primitive. Because only 20%–25% of the dynamically executed words are non-primitives, direct threading is faster on most processors, but not always on some popular processors (see Section 3).

The main disadvantage of direct threading in an implementation like Gforth is that it requires architecture-specific code for creating the contents of the code fields (Gforth currently supports direct threading on 7 architectures).

2.3 Primitive-centric threaded code

An alternative method to provide the body address is to simply lay it down into the threaded code as immediate parameter. Then the threaded code for a non-primitive does not point to the "code field" of the nonprimitive, but to a primitive that gets the inline parameter and performs the function of the non-primitive. For the variable `x` in our example this primitive is `lit` (the run-time primitive for `literal`).

This scheme is shown in Figure 3. The non-primitives still have a code field, which is not used in ordinary threaded code execution. But it is used for `execute` (and dodefer), because `execute` consumes a one-cell execution token (represented by a CFA), not a primitive with an argument.

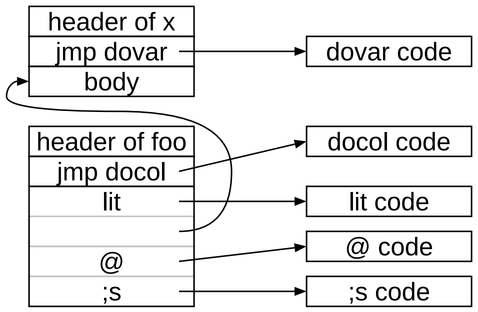

Figure 3: Primitive-centric direct threaded code

class	code
colon definition	`call` *`body`*
constant	`lit` *`value`*
value	`lit` *`body`* `@`
variable	`lit` *`body`*
user variable	`useraddr` *`offset`*
defered	`lit` *`body`* `@ execute`
field	`lit offset +`
does-defined	`lit` *`body`* `call` *`does-code`*
;code-defined	`lit` *`cfa`* `execute`

Figure 4: Compiling non-primitives for the primitive-centric scheme (with the least number of additional primitives)

Figure 4 shows how the various classes of non-primitives can be compiled. Instead of using a sequence of several primitives for some classes, new primitives could be introduced that perform the whole action in one primitive. However, if we combine frequent sequences of primitives into superinstructions, this will happen automatically; explicitly introducing these primitives may actually degrade the efficacy of the superinstruction optimization, because the optimizer would not know without additional effort that, e.g., our value-primitive is the same as the `lit @` combination it found elsewhere.

This scheme takes more space for the code of non-primitives; this is the original reason for preferring the traditional style when memory is scarce. There probably is little difference from traditional-style direct threading in run-time performance on most processors: Only non-primitives are affected; in the traditional-style scheme there is a jump and an addition to compute the body address, whereas in the primitive-centric scheme there is a load with often fully-exposed latency. For some popular processors the performance difference can be large, though (see Section 3).

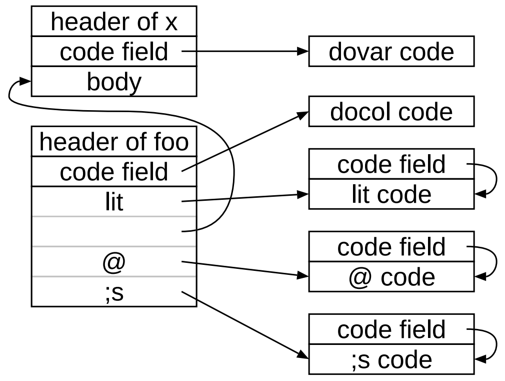

Figure 5: Hybrid direct/indirect threaded code

2.4 Hybrid direct/indirect threaded code

With the primitive-centric scheme the Forth engine (inner interpreter, primitives, and doers) is divided into two parts that are relatively independent of each other:

- Ordinary threaded code.
- `Execute` (and dodefer), code fields and execution tokens.

Only `execute` (and dodefer) deals with execution tokens and code fields[1], so we can modify these components in coordinated ways. The modification we are interested in is to use indirect threaded code fields; of course this requires an additional indirection in the `execute` code, it requires adding code fields to primitives (see Fig. 5), and the execution tokens are the addresses of the code fields. However, the ordinary threaded code points directly to the code of the primitives, and uses direct threaded NEXTs.

The main advantage of this hybrid scheme is that we do not need to create machine-code jumps in the code field, which helps the portability goals of Gforth and reduces the maintenance effort. Another advantage is that this allows us to separate code and data completely, without incurring the run-time cost of indirect threaded code for most of the code.

The overall performance impact should be small, because `execute` and dodefer account for only 1%–1.6% of the executed primitives and doers; and it should be a slight speedup on most processors, because most (70%–97%) of the `execute`d or deferred words are colon definitions, and indirect threaded code is often faster for non-primitives (because a jump is often more expensive than a `@`).

[1]Expanding ;code-defined words into `lit` *cfa* `execute` ensures that this holds for uses of ;code-defined words, too.

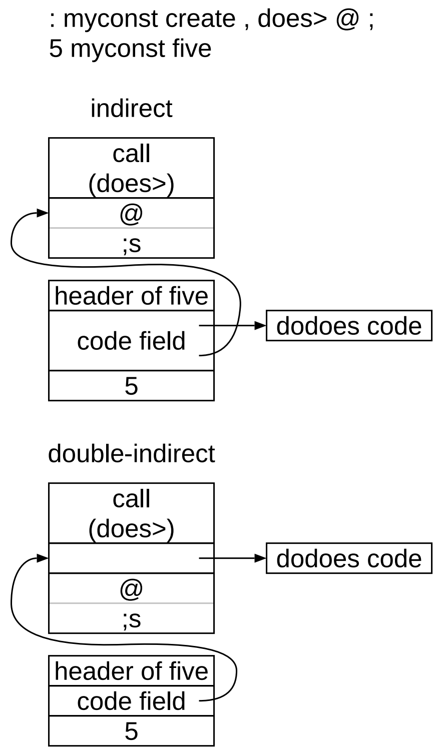

Figure 6: A `does>`-defined word with indirect and double-indirect threaded code.

Another interesting consequence of the division between ordinary threaded code and `execute` etc. in primitive-centric code is that doers (e.g., docol) can only be invoked by `execute`; therefore only `execute` has to remember the CFA for use by the doers. Unfortunately, the obvious way of expressing this in GNU C leads to the conservative assumption that this value is alive (needs to be preserved) across all primitives, and this results in suboptimal register allocation.

2.5 Double-indirect threaded code

For `does>`-defined words, the inner interpreter gets the CFA and has to find the body, the code address of dodoes, and the does-code (the Forth code behind the `does>`). With indirect threaded code, Gforth uses a two-cell code field that contains the code address and the does-code. Therefore all code fields in Gforth are two cells wide.

An alternative would be to use a double indirection (`@ @`) to get from the code field to the code address (double-indirect threaded code, see Fig. 6). In this scheme, the code field for `does>`-defined words points near the does-code, and the cell there points to dodoes.

This allows to reduce the code field size to one cell, but requires an additional indirection on every use of the code field; using double-indirect thread-

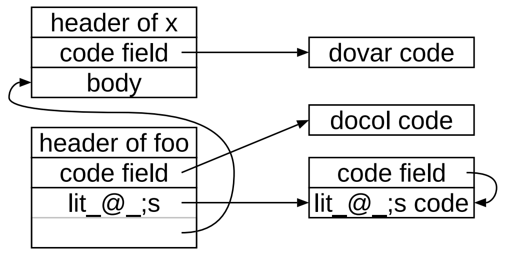

Figure 7: Combining primitives into superinstructions in hybrid direct/indirect threaded code

ing for all code would incur a significant performance penalty, but the penalty is small for a hybrid with a primitive-centric direct threading scheme. Double-indirect threaded code also requires one additional cell for the additional indirection for every primitive.

Forth systems using indirect threaded code and one-cell code fields usually use a scheme for implementing `does>`-defined words that is similar to our double-indirect threaded scheme, with one difference: instead of a pointer to dodoes there is a machine-code jump to dodoes between `(does>)` and the does-code. Gforth uses an appropriately modified variant of this approach for direct threaded code on some platforms [Ert93].

2.6 Superinstructions

We can add new primitives (superinstructions) that perform the action of a sequence of primitives. These superinstructions are then used in place of the original sequence; the immediate parameters to the original primitives are just appended to the superinstruction (see Fig. 7). In Gforth we select a few hundred frequently occuring sequences as superinstructions.

This optimization can be used with any of the threading schemes explained earlier, but it gives better results with the primitive-centric schemes, because a traditional-style non-primitive cannot be part of a superinstruction. However, it is possible to use the traditional scheme and convert only the invocations of those non-primitives into primitive-centric style that are to be included in superinstructions.

Superinstructions reduce the threaded code size, but require more native code. The main advantage (and the reason for their implementation in Gforth) is the reduction in run-time, mainly by reducing the number of mispredicted indirect branches.

3 Cache consistency

This section discusses an issue that has a significant performance impact on most Forth implementation techniques on the 386 architecture.

Modern CPUs usually split the first-level cache into an instruction cache and a data cache. Instruction fetches access the instruction cache, loads and stores access the data cache.

If a store writes an instruction, how does the instruction cache learn about that? On most architectures the program is required to announce this to the CPU in some special way after writing the instruction, and before executing it. However, the 386 architecture does not require any such software support[2], so the hardware has to deal with this problem by itself. The various implementations of this architecture deal with this problem in the following ways:

- The Pentium, Pentium MMX, K5, K6, K6-2, and K6-3 don't allow a cache line (32 bytes on these processors) to be in both the instruction and the data cache. If the instruction cache loads the line, the data cache has to evict it first, writing back the modified instruction. Similarly, when the line is loaded into the data cache, it is invalidated in the instruction cache. The effect of this is that having alternating data accesses and instruction executions in the same cache line is relatively expensive (dozens of cycles on each switch).

- The Pentium Pro, Pentium II, Pentium III, Celeron, Athlon and Duron allow the same cache line to be in both caches, in shared state (i.e., read-only). As soon as there is a write to the line, the line is evicted from the instruction cache. The effect is that it is relatively expensive to alternate writes to and instruction executions from the same cache line (32 bytes on the Intel processors, 64 bytes on the AMDs).

- According to the Pentium 4 optimization manual [Int01], it is expensive on the Pentium 4 to alternate writes and instruction executions from the same 1KB-region; actually the manual recommends keeping code and data on separate pages (4KB).

How are various Forth implementation techniques affected by that?

Indirect-threaded code. If primitives are implemented as shown in Fig. 1, i.e., with code fields adjacent to the code, the data read from the code field is soon followed by an instruction read nearby (usually in the same cache line). This leads to low performance on Pentium–K6-3; e.g., Win32Forth suffered heavily from this.

[2]Actually, the 486 requires a jump in order to flush the pipeline, but that does not help with the cache consistency problem.

Fortunately, this problem is easy to avoid by putting the code into an area separate from the code fields. Gforth does this, and performs well on these processors with indirect threaded code.

Direct-threaded code. Non-primitives have a jump in the code field close to the data that usually is accessed soon after executing the jump. So for traditional-style direct-threaded code the Pentium–K6-3 will slow down when executing non-primitives, and the Pentium Pro–Pentium 4 will slow down when writing to variables and values. There may be additional slowdowns due to code fields for other words being in the same cache line as the data, especially with the longer cache lines of the Athlon/Duron, and the 1KB consistency checking region of the Pentium 4. Note that these problems do not show up when running some of the popular small benchmarks, because their inner loops often contain only primitives.

These problems can be avoided by using indirect threaded code or by using primitive-centric direct-threaded code; in the latter case `execute` still executes the jumps in the code fields and will cause slowdowns. This can be avoided by using a hybrid direct/indirect-threaded code scheme.

Native code. Many Forth-to-native-code compilers store the native code `here`, interleaving that code with data for variables, headers, etc. This can lead to performance loss due to cache consistency maintenance in a somewhat erratic fashion (depending on which data shares a cache line with which code in a particular run). It also leads to bad utilization of both caches.

This problem can be avoided by having separate code and data memory areas.

4 Evaluation

This section compares the execution speed and memory requirements of a number of variants of Gforth:

trad Traditional-style threaded code.

doprims Primitive-centric threaded code using special primitives for values, fields, etc., such that only one primitive is executed per non-primitive.

0 Primitive-centric code, expanding some words into multiple primitives, as shown in Fig. 4. Equivalent to using 0 superinstructions.

50, 100, 200, 400, 800, 1600 Using 50, 100, ... superinstructions representing the n most frequently executed sequences in *brainless* (a chess program written by David Kühling).

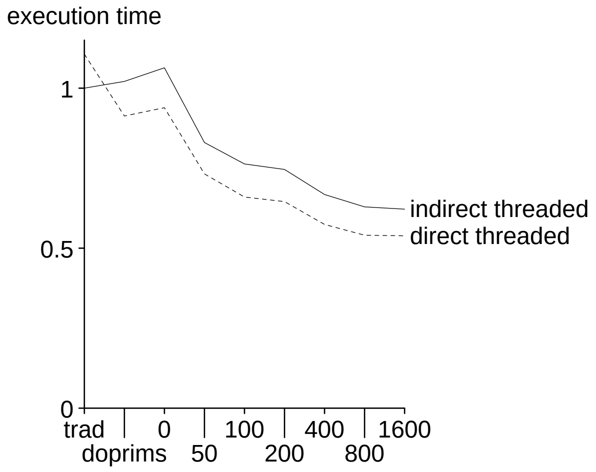

Figure 8: Execution time of bench-gc relative to traditional-style indirect threaded code on an 800MHz Athlon

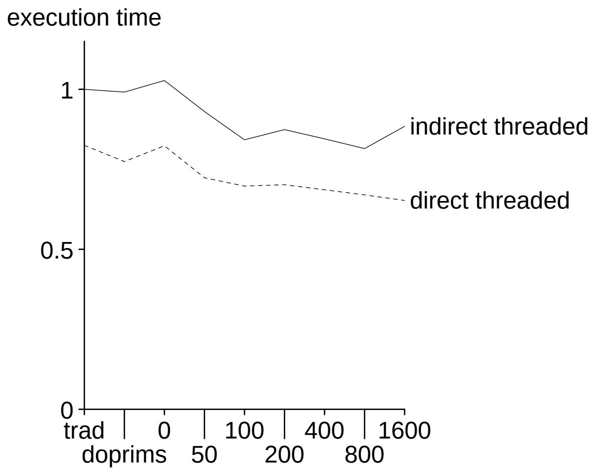

Figure 9: Execution time of bench-gc relative to traditional-style indirect threaded code on a 600MHz 21164a

We built and ran all of these variants with both indirect and direct threaded code, making it possible to identify how various components of a scheme contribute to performance. We did not run hybrid schemes (they are not yet implemented in Gforth), but we expect the performance to be very close to direct threaded code. For technical reasons, the kernel of Gforth (a part that contains, e.g., the compiler and text interpreter, but not the full wordlist and search order support) is always compiled in mostly Scheme 0 in these experiments.

4.1 Speed

Figure 8 and 9 show how the *bench-gc* benchmark behaves on an Athlon and a 21164a (Alpha architecture), respectively. Other benchmarks behave in similar ways, although the magnitude of the effects varies with the benchmark and with the processor.

For the traditional-style scheme, indirect threading is faster than direct threading on current 386 architecture processors (by up to a factor of 3), because of cache consistency issues; on other architectures direct threading beats indirect threading also for the traditional-style scheme, but not by as much as in the other schemes (because of the additional jumps).

The primitive-centric scheme gives about the same performance as the traditional-style scheme for indirect threaded code. For direct threaded code it produces a good speedup on the 386 architecture, mainly due to the elimination of cache invalidations. On the Alpha it gets a little faster, because of the elimination of the jumps through the code fields. Overall, the combination of the primitive-centric scheme and direct threading runs faster than traditional indirect threaded code on all processors.

Expanding non-primitives into multiple primitives (Scheme 0) costs some performance, but introducing superinstructions recovers that right away. On the Athlon, the Pentium III, and the 21264 the speedup from superinstructions is typically up to a factor of 2 for large benchmarks, and up to a factor of 5.6 on small benchmarks (*matrix*). A large part of this speedup is caused by the improved indirect branch prediction accuracy of the BTB on these CPUs. The speedup from superinstructions is quite a bit less on processors without BTB like the 21164a and the K6-2 (e.g., for bench-gc a factor of 1.38 on the K6-2 vs. 1.86 for the Athlon).

Having a large number of superinstructions gives diminishing returns. On the 21164a, there are also slowdowns from more superinstructions in some configurations; this is probably due to conflict misses in the small (8KB) direct-mapped instruction cache.

Using a large number of superinstructions also poses some practical problems: Compiling Gforth with 800 superinstructions requires about 100MB virtual memory on the 386 architecture, compiling it with 1600 superinstructions requires about 300MB and 1.5 hours on a 800MHz Athlon with 192MB RAM. So, in the release version Gforth will probably use only a few hundred superinstructions.

4.2 Size

Figure 10 shows data about the sizes of various components of Gforth:

Threaded code is the non-kernel part of the threaded code of Gforth for the 386 architecture; it contains, e.g., the full wordlist and search order support, `see`, and the assembler and disassembler. The shown size includes everything that appears in the dictionary, not just the threaded code. The code grows quite a bit by going to the primitive-centric scheme, and the savings incurred by superinstructions cannot fully recover that cost.

Native code is the text size of the Gforth binary, which consists mainly of the code for the primitives. For large numbers of superinstructions, this grows significantly.

Engine data is the data size of the Gforth binary, which contains superinstruction optimization tables, among other things. These, of course grow with the number of superinstructions.

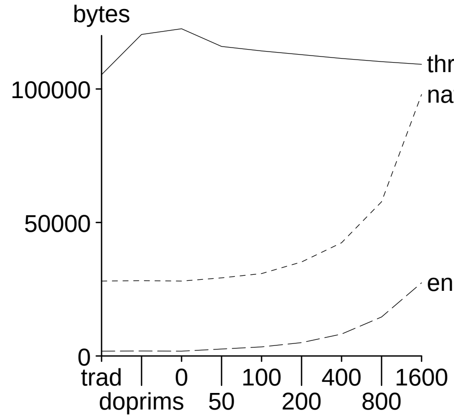

Figure 10: The sizes of Gforth's primitives (native code), engine data (mainly peephole optimization tables), and threaded code affected by the different schemes

Overall, superinstructions as currently implemented in Gforth cannot reduce the code size; other techniques, like converting only those non-primitives into a primitive-centric form that are combined into a superinstruction, or eliminating code fields, and switching to byte code might change the picture for sufficiently large programs. On the other hand, the increase in memory consumption from primitive-centric schemes and superinstructions should not be a problem in desktop and server systems. But for many embedded systems, the traditional scheme will continue to be the threaded-code method of choice.

5 Conclusion

Switching from traditional-style threaded code to a primitive-centric scheme opens up a number of opportunities, most notably a wider applicability of superinstructions, but it also makes direct threaded code viable on 386 architecture processors, and enables a hybrid direct/indirect (or direct/double-indirect) threading scheme that offers the performance of direct-threaded code without requiring any (non-portable) machine-code generation. The price paid is a moderate increase in threaded-code size.

Superinstructions offer good speedups on processors with BTBs, moderate speedups on processors without BTBs, and a moderate reduction in threaded code size (but not enough to fully recover the increase through primitive-centric code), but require more space for the primitives.

References

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

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

[Hug82] R. J. M. Hughes. Super-combinators. In *Conference Record of the 1980 LISP Conference, Stanford, CA*, pages 1–11, New York, 1982. ACM.

[Int01] Intel. *Intel Pentium 4 Processor Optimization*, 2001.

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

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