Implementation Issues for Superinstructions in Gforth

M. Anton Ertl*
TU Wien

David Gregg
Trinity College Dublin

Abstract

Combining Forth primitives into superinstructions provides nice speedups. Several approaches to superinstructions were explored in the Gforth project. This paper discusses the effects of these approaches on performance, compilation time, implementation effort, and on programming tools such as the decompiler and backtracing.

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. Superinstructions are also used as the main optimization method in SwiftForth (a native-code compiler).

This paper covers all the work we have done on superinstructions in Gforth. Some aspects have been discussed in detail elsewhere [EG01, EGKP02, Ert02, EG03], and are only mentioned superficially here, with a little new material here and there: Section 2 gives an overview of primitive-centric code and dynamic and static superinstructions. Section 3 looks at the basic performance effects of superinstructions. Section 5 discusses the related work.

Other aspects are discussed here for the first time, including different ways of selecting static superinstructions (Section 4.2) and the interaction with dynamic superinstructions (Section 4.4), image file format issues (Section 4.5 and 4.6) and decompilation (Section 4.8–4.10).

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

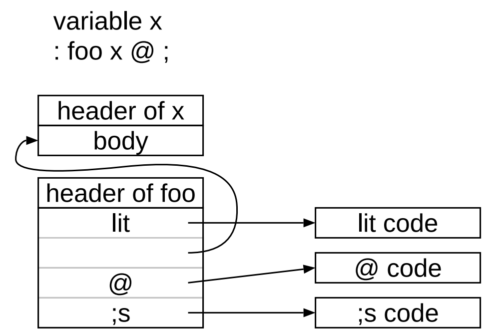

Figure 1: A piece of Forth code and its representation as primitive-centric direct threaded code

2 Superinstructions

This section explains the basic concepts of static and dynamic superinstructions.

2.1 Primitive-centric threaded code

Unless otherwise noted, we assume in this paper that we are using primitive-centric direct threaded code [Ert02]. I.e., threaded code where every use of a non-primitive (like colon definitions or variables) is compiled to a primitive with an immediate parameter; e.g. a reference to a variable is compiled to a `lit` primitive with the PFA (body address) of the variable as inline parameter (see Fig. 1).

The reason for using primitive-centric code instead of classical indirect threaded code (or some direct-threaded variant of it) is that we can combine all primitives into superinstructions, whereas we cannot combine a non-primitive in classical indirect-threaded code; e.g., consider the sequence `x @`, where `x` is a variable; if we wanted to combine this sequence into a superinstruction in classical code, we would need to have a CFA for `x @` in the head of `x` (in addition to the CFA for `x` alone, which may be needed elsewhere).

Alternatively, we could change `x` into primitive-centric code when creating the superinstruction, but that is both harder to explain (so we do not do it in this paper) and to implement (so we do not do it in Gforth).

2.2 Dynamic superinstructions

The simplest way to construct superinstructions is to concatenate the code of their component primitives,

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: EuroForth 2003 Conference Proceedings

```
ebx=IP  ecx=TOS  esi=SP  eax=tmp

no superinstructions      dynamic superinstructions   static superinstructions
Code lit                                              Code lit+
mov dword ptr [esi], ecx  mov dword ptr [esi], ecx    mov eax, dword ptr [ebx]
mov ecx, dword ptr [ebx]  mov ecx, dword ptr [ebx]    add ebx, # 8
add esi, # -4             add esi, # -4               add ecx, eax
add ebx, # 8              add ebx, # 8                jmp dword ptr -4[ebx]
jmp dword ptr -4[ebx]                                 end-code
end-code

Code +
mov eax, dword ptr 4[esi] mov eax, dword ptr 4[esi]
add esi, # 4              add esi, # 4
add ebx, # 4              add ebx, # 4
add ecx, eax              add ecx, eax
jmp dword ptr -4[ebx]     \ append next primitive
end-code
```

Figure 2: The native code of static and dynamic superinstructions

leaving away the dispatch jump (see Fig. 2, middle).

This way of constructing superinstructions is so easy that we can do it while the Forth system compiles the Forth program to threaded code, i.e., at run-time of the Forth system (dynamically), with very little machine-specific code:

We just have to set up a memory region for the native code of the superinstructions. When the Forth system compiles a primitive, it copies the native code of the currently-compiled primitive without the dispatch jump to the end of the current superinstruction. After each branching primitive (including, e.g., `;s` compiled by `exit`) a dispatch jump is appended. On the threaded-code side the Forth compiler compiles pointers to this dynamically generated native code instead of pointers to the original primitives.

This approach has been proposed by Piumarta and Riccardi [PR98], with improvements proposed by us [EG03]. Gforth 0.6 uses dynamic superinstructions by default on most architectures.

2.3 Static superinstructions

In the code of a dynamic superinstruction, there are a lot of optimization opportunities, in particular for the stack accesses: many stores to and loads from the stacks can be eliminated by keeping the stack item in a register, and several stack pointer updates for one stack can be combined into one update or even eliminated. E.g., consider the `lit+` superinstruction in Fig. 2: It eliminates the data stack store of `lit` and the data stack load of `+`, and eliminates the data stack pointer updates of both; as a result, this superinstruction is shorter than either `lit` or `+`.

Unfortunately, these optimizations cannot be performed at run-time with less effort (and lack of portability) than it would take to build an optimizing native-code compiler of the calibre of VFX [Pel98] (or, if we are content with optimizing only the simple cases, bigForth [Pay91]); and if we were willing invest that effort, there would be not much point to think in terms of superinstructions at all.

However, we can decide in advance that we want to use specific superinstructions, and optimize them. Then, at Forth system run-time, when compiling Forth code, use the optimized superinstruction whenever the appropriate sequence of primitives comes up.

Optimizing the superinstructions in advance is often done manually, e.g., in SwiftForth. Vmgen [EGKP02], the generator used to build Gforth's engine, supports optimizing superinstructions automatically, and also helps determining which sequences occur frequently, thus supporting the selection of profitable superinstructions.

Selecting a good set of static superinstructions is an interesting topic that we have explored earlier [GW02] and will not discuss further here.

Gforth 0.6.2 uses a set of 27 static superinstructions in the `gforth-fast` engine.

3 Basic Performance

In this section we look at the basic performance effects of various forms of superinstructions.

The performance effects of superinstructions depend very much on the microarchitecture of the processor, in particular on the cost of a branch misprediction, on the indirect branch predictor, and on the store bandwidth.

On most popular general-purpose processors today (e.g., the Pentium 4, the Athlon family, the Pentium III family, and the Alpha 21264 family) the cost of a branch misprediction is high (≥ 10 cycles), they have a branch target buffer (BTB) or similar feature for predicting indirect branches, and store bandwidth is not an issue in our context. We will concentrate on such processors in this section, because we have most experience and measurements with such processors.

Common properties of other processors are:

- Branch mispredictions are relatively expensive ($\geq$ 5 cycles) and they always mispredict indirect branches (most PowerPCs and the Alpha 21164).
- Branches are relatively cheap (2 cycles) and there is no branch prediction (R3000, StrongARM).
- The processor has limited store bandwidth due to a write-through L1 cache and limited bandwidth to outer layers of the memory hierarchy (microSPARC II, 486DX2/66, 21164PC, 21064a, XScale).

Such processors are often used in embedded systems. The following intermediate effects have very different performance effects on the different processor classes.

3.1 Indirect Branch Prediction

Processors with BTBs mispredict 50%–60% of the indirect branches for typical threaded-code interpreters [EG01]. Since the branch mispredict penalty is quite high on such processors, they can spend 60% of their time (for Athlon and Celeron on Gforth running benchgc) in branch mispredictions [EG03].

Nearly all of these mispredictions can be eliminated with a technique called dynamic replication [EG03], where each instance of a primitive in the threaded code gets its own copy of the native code for the primitive and, in particular, its dispatch indirect branch. This leads to speedups by up to a factor of 2.39 on a Celeron 800 [EG03].

Superinstructions also have a replication effect (they introduce more different dispatch branches), and this is their dominant performance effect on processors with BTBs.

In particular, dynamic superinstructions as implemented in Gforth perform dynamic replication as well (if the same dynamic superinstruction occurs twice, its native code is generated and used twice), and therefore have the same effect on indirect branch prediction as dynamic replication.

Static superinstructions alone have less replication effect, reducing the misprediction rate to 20%–30% [EG03]. As a result, static superinstructions alone provide less of a speedup than dynamic superinstructions alone, in spite of the reduction in executed instructions that static superinstructions provide.

However, static superinstructions can be combined with dynamic superinstructions (see Section 4.4) to gain all the branch prediction benefits of dynamic superinstructions and the other benefits of static superinstructions.

Of course, on processors without BTBs, there is no benefit to replication and the replicative effects of superinstructions.

3.2 Reducing indirect branches

The benefit of dynamic superinstructions (with replication) over dynamic replication alone is the reduction in dispatch branches (and maybe other parts of NEXT). On processors with BTBs correctly predicted branches are relatively cheap (e.g., 2.2 cycles or less on the Athlon), so the benefit of this optimization is relatively small on such processors (a factor 1.18–1.32 on the Celeron, for a total speedup from dynamic superinstructions over ordinary threaded code of a factor of up to 3.09 [EG03]).

On processors without a BTB and relatively expensive indirect branches this benefit is higher: factor 1.36–1.61 on a PPC604e, 1.30-1.52 on a 21164a.

Dynamic superinstructions always eliminate all but one of the dispatch branches within a basic block (straight-line code sequence). This typically reduces the number of executed dispatch branches by a factor of about 3 on Gforth code; optimizations that increase the basic block length (in particular, inlining) would enhance this effect.

By contrast, static superinstructions alone reduce the number of dispatch branches only by about a factor of about 1.5. Again, the combination of dynamic and static superinstructions offers the same reduction in dispatch branches as dynamic superinstructions alone, while gaining all of the other benefits of static superinstructions.

Reducing dispatch branches has no performance effect on processors that are store bandwidth limited most or all of the time (e.g. MicroSPARC II).

3.3 Reducing other instructions

The main benefit of static superinstructions over dynamic superinstructions is the reduction in stack loads, stack stores, and stack pointer updates. In some cases, additional optimizations are possible (e.g., combining the main computation of `cells + @` into one instruction on the 386 architecture).

We can isolate this benefit by comparing dynamic superinstructions alone with the combination of dyamic and static superinstructions. Both variants have the same behaviour with respect to branch prediction and the number of dispatch branches, so the only differences are the optimizations that static superinstructions have. Also, these two variants are the two variants we would commonly expect to be used on general-purpose machines.

The results on the Athlon are relatively disappointing. For a realistic selection of superinstructions we see only speedups by factors around 1.1.

We then checked what is possible in the best case (better than realistically possible), and selected superinstructions to cover the 300 most frequently executed basic blocks of the benchmark *brainless*, and then used them on a run of the same benchmark. As a result, we saw speedups of 1.22 on a Pentium 4, 1.25 on a Pentium III, and 1.20 on an Athlon. Performance counters measurements revealed a reduction by a factor of 1.6 in the number of executed (retired) instructions. Further investigations with the performance counters did not turn up a good explanation yet why the speedup is so much lower than the reduction in the number of executed instructions.

On other kinds of processors the speedups may be better. In particular, we expect better speedups on simple processors with more uniform instruction execution times like the R3000 and the StrongARM, and on processors that are store bandwidth limited, like the microSPARC II. However, we have yet to perform these measurements.

3.4 Instruction cache misses

Many people expect that instruction cache misses become a problem with dynamic dynamic superinstructions as implemented in Gforth (with replication). In our benchmarking only on one benchmark (brainless) on the Celeron-800 (but not on the Athlon) the effect of the increase in I-cache misses was larger than the effect of the reduction in branch mispredictions [EG03]. We think that I-cache misses are not a big problem for most applications.

One way to think about this is that the native code produced by dynamic superinstructions is similar to the code produced by a native code compiler (larger only by a small constant factor). If I-cache misses are not a big issue for native-code compilers, they are not a big issue for dynamic superinstructions, either.

3.5 Compile time

Some people worry about the cost of the code copying taking place with dynamic superinstructions. When loading the Gforth image for the 386 (containing about 15000 words of threaded code (plus immediate arguments, headers, etc.), with about 200KB of native code generated, this takes about 5 ms on a Celeron-800. Moreover, the speedup over plain threaded code amortizes this overhead already during the Forth-level startup code, leading to the same total startup times (17 ms on a Celeron-800).[EG03]

4 Details

4.1 Threaded-code slots

In the original proposals for both dynamic superinstructions [PR98] and for static superinstructions [EGKP02] there were no threaded-code slots for optimized-away primitives (see Fig. 3, left). This was also the case for the first implementation of static superinstructions in Gforth.

However, the dynamic superinstruction implementation in Gforth and the new static superinstruction implementation leave slots for these primitives (Fig. 3, right). This has a number of advantages, and a few minor disadvantages. Most of them are discussed in subsequent sections; here we will discuss only the threaded-code size issue.

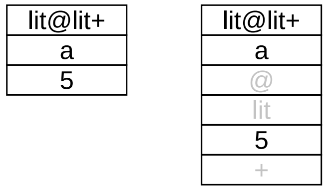

Figure 3: Threaded code for the superinstruction for `a @ 5 +` without and with slots for the optimized-away primitives

The most obvious disadvantage is that superinstructions no longer reduce the threaded-code size. How much threaded code would we save? The Gforth image for the 386 architecture contains 181KB of Forth code and data. Of that, 60KB is in threaded code words (without inline parameters). By eliminating the unused slots of static superinstructions, we could eliminate about 20KB (depending on the set of static superinstructions). By eliminating the unused slots of dynamic superinstructions, we could eliminate about 40KB of threaded code.

4.2 Static superinstruction selection variants

The first implementation of static superinstructions used a very simple greedy peephole optimization algorithm for selecting superinstructions: It always looked if the current primitive can be combined with the last one compiled. If it could, the last one was updated in place, otherwise the current one was compiled with `,`.

For longer superinstructions, this method requires that all prefixes are present. I.e., `lit@lit+` would require the prefixes `lit@lit` and `lit@`, otherwise it would never optimize a sequence into `lit@lit+`.

Another disadvantage of this method is that it may miss the optimal solution. E.g., if there are superinstructions `AB` and `BCDEF` (and its prefixes), then the greedy method will compile `A B C D E F` into `AB C D E F`, whereas `A BCDEF` would probably be better.

It is possible to eliminate the prefix requirement and use optimal superinstruction selection with various metrics for optimality, such as minimum native code size, minimum number of loads, stores and stack pointer updates, or minimum number of static superinstructions. This superinstruction selection algorithm uses dynamic programming[BCW90], as usual in shortest-path algorithms:

We start by compiling the code in the usual way (without any superinstruction selection), but in addition we record the address of each threaded-code word. When we reach the end of a basic block (i.e., some control-flow word, e.g., a call), we compute the optimal solution (see below), then rewrite the threaded-code words in the basic block to use the instructions in the optimal solution. Since the slots for the optimized-

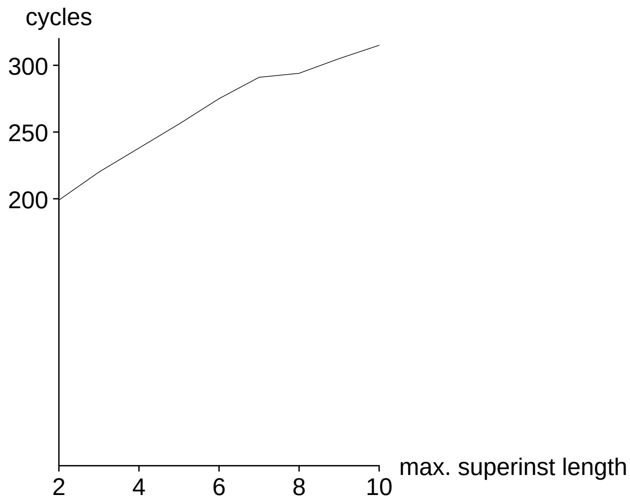

Figure 4: Additional compile time per primitive for optimal superinstruction selection on an Athlon-1200

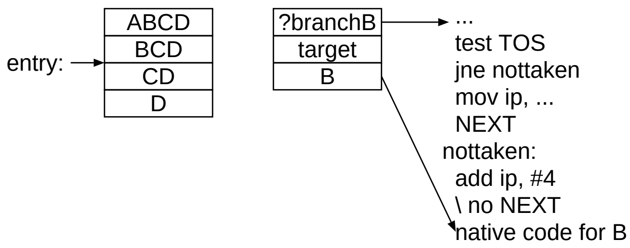

Figure 5: Superinstructions across entry points and across conditional branches

away instructions stay where they are, we can leave the immediate arguments in the code just where they are.

Computing the optimal solution starts from the end of the basic block, goes backwards, and always computes the best solution from the current position to the end of the basic block. At each position it looks at all the superinstructions possible in this position, and computes which of them is cheapest when you add the cost of the superinstruction and the optimal cost from the end of the superinstruction to the end of the basic block (we have computed these costs before). This algorithm takes $O(nm)$ time, where n is the length of the basic block, and m is the length of the longest superinstruction.

Preliminary results indicate that this optimal superinstruction selection does not buy a significant speedup over the greedy scheme, at least for the sets of superinstructions that we have tried.

Given that the performance effect of this optimal algorithm is small, what are its costs? On a 1200MHz Athlon, it costs 199–315 cycles per compiled primitive, depending on the length of the longest superinstruction, see Fig. 4; the line flattens for longer superinstructions because there are fewer basic blocks of that length. The total additional cost is 2.5ms–4ms when loading the Gforth image. This code is not very optimized, and could be sped up significantly, e.g., by using an automaton for the shortest-path search [Pro95a].

4.3 Superinstructions across basic blocks

There is no reason to limit superinstructions to basic blocks. There are two kinds of basic block boundaries:

Entry points: We have slots for all original primitives, and they are filled with correct code. Therefore, any slot is a correct entry point, even if it is skipped by some superinstructions (see Fig. 5, left).

Branches: Unconditional branches end a superinstruction, because dealing with threaded-code control flow changes within a superinstruction would be too complicated. But for the not-taken path of a conditional branch primitive the superinstruction can continue. For dynamic superinstructions the native code `?branch` (and other conditional branch primitives) must be arranged such that the dispatch branch of the fall-through path through the primitive is last. Then it can be left away to continue the dynamic superinstruction (see Fig. 5, right).

Gforth 0.6.2 supports static and dynamic superinstructions across entry points, and dynamic superinstructions across conditional branches.

The speedups from this improvement are relatively small (up to 1.11 for brainless on the Celeron-800 [EG03]). The reason for this is that most dynamically occuring basic block boundaries in Forth are calls, returns, and the corresponding entry and return points; conditional branches and entry points joining another path are only a minority.

One benefit of being able to use superinstructions across entry points is that it simplifies the image loader: We do not need to scan the images for entry points.

4.4 Combining static and dynamic superinstructions

Implementing only dynamic superinstructions is easy: just append the native code of the currently compiled primitive at the end of the native code region, sometimes without, sometimes with the dispatch branch at the end.

The addition of static superinstructions complicates this slightly. Static superinstructions skip some instruction slots during execution, and this should be reflected during dynamic superinstruction generation. Dynamic superinstruction generation is performed during the rewriting step of static superinstruction selection. We just keep track of which slot is the next one that should not be skipped.

As a result, in the end the threaded code contains a mixture of references to statically and dynamically

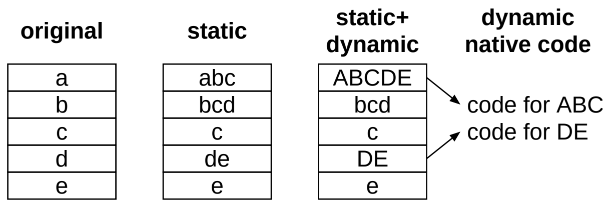

Figure 6: Superinstructions generated by combining static and dynamic superinstructions. Case distinguishes references to statically and dynamically generated native code.

generated native code (see Fig. 6). If you start executing the sequence at the start, you will only execute dynamically generated code (with its advantages in branch prediction and dispatch branch elimination). However, if you enter the sequence somewhere else, you might execute one or more statically generated superinstructions or primitives before hitting dynamically generated native code.

One way to avoid this would be not to allow static superinstructions across entry points. This would allow us to gain more of the advantages of dynamic superinstructions, but foregoes some of the advantages of static superinstructions.

Currently we have implemented only one variant (static superinstructions across entry points, because there is no easy way to determine all entry points in an image), so we cannot say which variant performs better.

4.5 Image format

One of the goals in Gforth development is to have a relatively stable image file format. If superinstructions were represented in the image file, we would have a much harder time to adapt the set of static superinstructions in the engine to new insights or to the individual capabilities of the platform (in particular, the amount of RAM for compiling the engine).

The first implementation of static superinstructions required representing superinstructions in the image file: there was only one threaded code slot for the superinstruction, so we needed to encode the whole superinstruction there, or information would be lost.

In contrast, in the current implementation (0.6.2), we can encode the original primitives in the image file, and combine them into superinstructions at image load time.

4.6 Image generation

A related issue is the way the images are generated. There are two ways of generating Gforth images: using the cross compiler (mainly used for building a kernel), and using the Forth compiler (used for extending the system beyond the kernel).

If superinstructions were represented in the image, both compilers would have to be extended to generate superinstructions in order to make full use of them. In the first implementation, only the Forth compiler was extended, so superinstructions were not used in the kernel.

In the Gforth 0.6.2 the superinstruction generation (for the static and dynamic superinstructions) is implemented only once, and called by both the image loader and the Forth compiler; the cross compiler does not have the slightest idea if and what kind of superinstructions will be used for the images it generates.

A minor disadvantage (for Forth fans) of this design is that superinstruction generation is not implemented in Forth, because it is used by the image loader, and that runs before Forth code can be run.

4.7 Superinstruction primitives

There are some primitives in Gforth that would serve nicely as static superinstructions: These primitives include `lit+` and `lit@` used by the cross compiler to implement non-primitives in a uniform way (one primitive with one immediate argument); and words like `?dup-?branch`, `?dup-0=-?branch`, `under+`, `nip`, `tuck`, `-rot` etc., where users often use equivalent sequences of primitives.

Unfortunately, these primitives cannot be used as static superinstructions in the current implementation: the superinstruction implementation leaves slots for the component primitives, but these primitives do not expect such slots.

So these primitives are not used as superinstructions. Instead, there are superinstructions for some of the equivalent sequences (e.g., `lit @`). A consequence of this is that a static superinstruction containing either `lit@` or `lit @` can be applied in fewer cases, or equivalently, we need more static superinstructions to achieve the same amount of optimization. This can be partially addressed by changing at least the Forth compiler to generate `lit @` instead of `lit@` (then `lit@` occurs only in code produced by the cross compiler). We have not yet evaluated the effect of this change.

Another way to address this problem would be to put a small greedy peephole optimizer into `compile,` that combines such sequences of primitives into equivalent primitives before passing them on to static and dynamic superinstruction generation.

Alternatively, these words could be expanded to the equivalent sequences by `compile,`, and static superinstruction selection would combine them again (possibly with other primitives). Once we integrate an inliner into Gforth, we would get this solution for free.

In conclusion, keeping the original primitive slots has some disadvantages, but overall the advantages outweigh the disadvantages.

4.8 Decompiler

The decompiler is a variation of the original threaded-code decompiler. In its original form, it steps through the threaded code, looks at the code addresses present there, looks up the corresponding primitive name and prints it (for `simple-see`), or does additional processing to recover the control structure and the non-primitives (for `see`).

For accommodating superinstructions, the decompiler stays essentially the same, except for a few well-placed calls to `decompile-prim`, which takes the code address pointing to a superinstruction and returns the code address of the first primitive in the superinstruction.

Internally, `decompile-prim` takes a look at the start of the code pointed to by the code address, and searches among the primitives and the static superinstructions for the longest match[1]. If `decompile-prim` finds a primitive, it just returns its code address; if it finds a static superinstruction, it returns the code address of the first component primitive of the static superinstruction.

Here, keeping the original primitive slots provides a significant advantage: Without them, the decompiler would have to know about superinstructions and keep track of how much of a superinstruction it has already processed.

4.9 Backtracer

If the Forth program produces an uncaught exception, Gforth prints a backtrace. The backtrace is generated by recording the return stack when the first exception happens. For exceptions produced by primitives (e.g., accessing the wrong address with `@`), the engine pushes the current IP (instruction pointer) on the return stack before throwing the exception. The backtracer then produces the name of the primitive just like the decompiler, using `decompile-prim`.

One complication is that, with static superinstructions, the current IP is not precisely known, as the exception may have happened at any of the components of the static superinstruction.

Our approach to this problem is to use static superinstructions only in the `gforth-fast` engine, which cannot produce full backtraces for exceptions caused by primitives anyway. In the `gforth` engine for debugging and for programs that are not time critical, we use only dynamic superinstructions; since only the dispatch jumps are optimized away by dynamic superinstructions in Gforth, each component primitive still records the IP when it starts executing.

Moreover, optimizing away stack accesses as happens in static superinstructions would be of questionable utility for the debugging engine, as it might cover up some stack overflow errors. We have already disabled keeping the TOS in a register in the `gforth` engine in order to catch all stack underflows. Therefore we might just as well go a little slower again to catch all stack overflows right when they happen.

On the other hand, stack overflows are less likely to be missed completely, and more likely to occur at some other place than the cause of the error; and in the absense of TOS caching, static superinstructions might provide more speedup.

4.10 Debugger

While many people advocate other debugging methods, some users still like to have stepping or tracing debuggers. In Gforth there is such a debugger that is based on patching the threaded code.[2] Here are the issues that such a debugger has in the presence of superinstructions:

Since such a debugger works by potentially patching the threaded code of each executed primitive, it requires that all the dispatches through the threaded code are actually performed. This means that there must be no superinstructions at all. Gforth actually provides such options (`--no-super` for disabling dynamic superinstructions, `--ss-number=0` for disabling static superinstructions), which still gives the user dynamic replication, which provides most of the speedup on processors with BTBs anyway.

Alternatively, the debugger could be modified to work by patching only threaded code that is guaranteed to be used in a dispatch; i.e. all the branch targets in a sequence of code. The debugger could show each sequence as one step, or use other methods to provide stepping within the sequence.

5 Related work

Superinstructions have been used in interpreters for a long time without leaving much trace in the published literature. The first publication we are aware of is on the related but more complex topic of superoperators [Hug82] in the context of combinator graph reduction for implementing lazy functional languages.

In the Forth community, the first publication we are aware of is by Schütz [Sch92]. Wil Baden [Bad95] also proposes using superinstructions, with a simple, greedy peephole optimization (called pinhole optimization by Baden).

Proebsting proposes a related scheme called superoperators for a C interpreter [Pro95b]. With superoperators a part of an expression tree is combined, instead of a sequence of instructions. Both schemes can do things that the other scheme cannot do, so it is unclear which one performs better.

Rossi and Sivalingam sketch dynamic superinstructions [RS96]. Piumarta and Riccardi discuss and evaluate them in more detail [PR98].

[1] If it just searches for any match, this could match a primitive that is a prefix of some other primitive (depending on the placement of the IP update), so searching for the longest match is safer. E.g., on the 386 architecture the code for `lit` is a prefix of the code for `lit@`.

[2] In Gforth 0.6, this debugger does not work, because it has not been kept up to date with the changes since 0.5.0 like primitive-centric threaded code and branches with absolute target addresses.

We propose automatic generation of optimized static superinstructions, originally with greedy superinstruction selection [EGKP02]. Gregg and Waldron evaluated strategies for selecting superinstructions from profile runs [GW02]. Primitive-centric threaded code is an enabling technology for using superinstructions [Ert02] (that paper also contains some superinstruction results).

Recently, we evaluated various techniques for reducing indirect branch mispredictions [EG03], among them dynamic and static superinstructions and their combination, providing the first empirical comparison of these techniques.

A classical method of generating native code for Forth is to inline the primitives [Ros86]; this corresponds to using dynamic superinstructions. SwiftForth combines this with using static superinstructions. The difference between a threaded-code interpreter employing dynamic (and maybe static) superinstructions and such a native-code compiler is that the interpreter still takes all the immediate arguments of the primitives (e.g., the target address of a call) from the threaded code, whereas the native-code compiler usually compiles these arguments directly into the native code.

6 Conclusion

Superinstructions can speed up Forth interpreters. For modern desktop processors, the biggest speedup comes from improved indirect branch prediction, resulting in a total speedup of up to a factor of 3.17 (a factor of 2 is more typical, though).

There are a number of implementation issues and design decisions when dealing with superinstructions that we have looked at in this paper: Selecting static superinstructions for a given sequence of primitives, where to end superinstructions, combining static and dynamic superinstructions, image format and cross-compilation issues, decompilation, backtracing, and debugging.

One apparently minor design decision is to keep slots for the original primitives in the threaded code for the superinstruction; this is helpful for most of the other implementation issues (static superinstruction selection, extending superinstructions across entry points, image format stability and cross-compiler simplicity, decompilation, and backtracing) and has only a few drawbacks (larger threaded-code size, and the inability to use existing primitives as superinstructions).

References

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

[BCW90] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice-Hall, 1990.

[EG01] M. Anton Ertl and David Gregg. The behaviour of efficient virtual machine interpreters on modern architectures. In *Euro-Par 2001*, pages 403–412. Springer LNCS 2150, 2001.

[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.

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

[Ert02] M. Anton Ertl. Threaded code variations and optimizations (extended version). In *Forth-Tagung 2002*, Garmisch-Partenkirchen, 2002.

[GW02] David Gregg and John Waldron. Primitive sequences in general purpose Forth programs. In M. Anton Ertl, editor, *18th EuroForth Conference*, pages 24–32, 2002. Refereed.

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

[Pay91] Bernd Paysan. Ein optimierender Forth-Compiler. *Vierte Dimension*, 7(3):22–25, September 1991.

[Pel98] Stephen Pelc. The MPE VFX Forth code generator. In *EuroForth '98*, 1998.

[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.

[Pro95a] Todd A. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, May 1995.

[Pro95b] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.

[Ros86] Anthony Rose. Design of a fast 68000-based subroutine-threaded Forth with inline code

& an optimizer. *Journal of Forth Application and Research*, 4(2):285–288, 1986. 1986 Rochester Forth Conference.

[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.

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