Inlining in Gforth: Early Experiences

David Gregg
Trinity College Dublin

M. Anton Ertl*
TU Wien

Abstract

Many optimizations are easier or more effective for straight-line code (basic blocks). Straight-line code in Forth is limited mainly by calls and returns. Inlining eliminates calls and returns, which in turn makes the basic blocks longer, and increases the effectiveness of other optimizations. In this paper we present a first prototype implementation of ininlining for Gforth.

1 Introduction

Many code generation tasks and optimizations are relatively easy to perform on basic blocks (code sections with only one entry point and one exit), often with optimal results (with respect to some optimality criterion). Examples are register allocation, static stack caching [EG04a], and superinstruction selection [EG03]; all of these problems are relevant for Forth implementations, and can be solved optimally in linear time for basic blocks.

At optimization boundaries usually some overhead occurs (at least you don't see all of the optimization benefit); e.g., the register allocator has to arrange the register contents in a canonical way, stack caching has to perform a transition into the canonical stack state, and superinstructions cannot extend across the boundary.

Therefore, lots of effort has been expended on extending the optimization boundaries beyond basic blocks, to loops (loop-level optimization), to general control flow graphs in procedures (global optimization), across procedure boundaries (interprocedural optimization), or to the whole program (whole-program optimization).

However, these extended optimizations are often much more complicated, require more compile-time, and often cannot be solved optimally in acceptable time. E.g., global register allocation turns into graph colouring, an NP-hard problem; i.e., it requires exponential time for the optimal solution; what's worse, even a heuristic suboptimal solution requires quite a complex program, and takes significant time[1].

Therefore, Forth compilers with their need for compilation speed have not embraced optimizations beyond basic blocks. E.g., RAFTS [EP96] and (to our knowledge) VFX perform their optimizations (e.g., register allocation) only at the basic block level. Cforth [Alm86] and (to the best of our knowledge) iForth perform global register allocation of a kind by continuing the current allocation across branches and performing reconciliation at control flow joins (not necessarily in a close-to-optimal way).

Another reason why loop-level and global optimizations are not very useful for Forth is that the main cause of basic block boundaries in Forth is calls and `exits` (see Fig. 1).

Therefore, if we want to optimize Forth significantly beyond basic blocks, it seems that we have to perform interprocedural optimization, which is even harder, more complicated and slower than global optimization, and not as well researched.

Fortunately, there is an alternative: We can replace a call to a colon definition with the body of the called definition (*inlining*). This eliminates calls and returns, and their implied optimization boundaries.

2 Code generator overview

This section gives some background information on the Gforth code generator, which is mentioned throughout the paper.

Figure 2 shows the various parts of Gforth involved in code generation, and the data they are working on.

One important issue is that there are two paths to code generation that are united by calling `compile-prim1`: One is regular Forth compilation through `find` and `compile,`; the other path is image loading.

The image loader has to run before any piece of Forth code runs, so it cannot be written in Forth, and nothing it calls must be written in Forth. So if we wrote any optimizations in Forth, they could not be applied at load time.

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

[1] Preliminary results for a JVM JIT compiler indicated 8000 cycles per bytecode (personal mail from Cliff Click).

calls	`exits`	`execute`	`BRANCH`	cond. branches	Source
12.21%	11.74%	0.65%	6.34%	1.54%	[Koo89]
13.46%	13.46%	1.24%	5.86%	0.93%	http://www.complang.tuwien.ac.at/forth/peep/sorted

Figure 1: Dynamic execution frequencies of control flow primitives

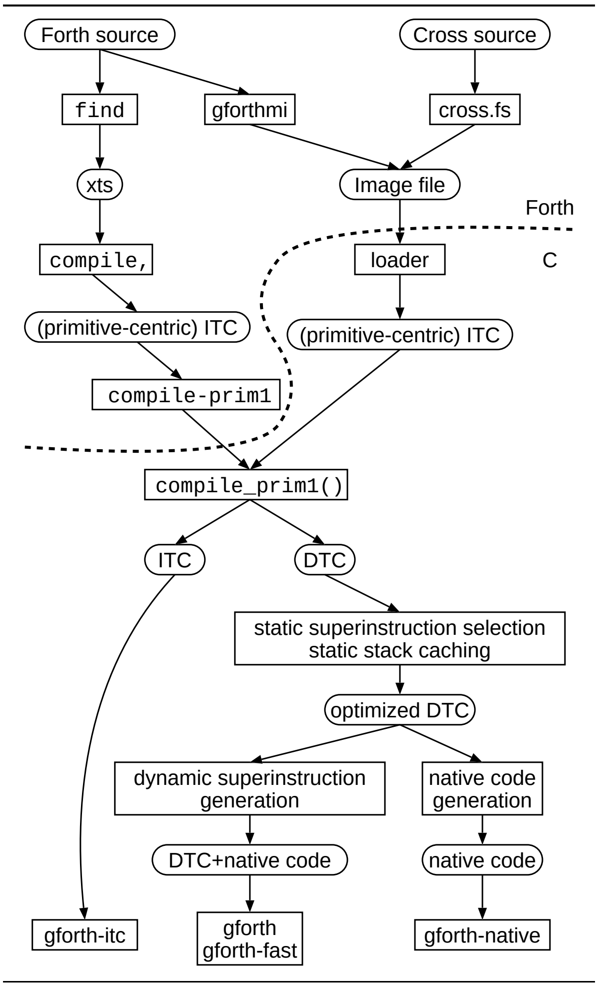


Figure 2: Code generation in Gforth

It might be possible to apply optimizations before image generation (in gforthmi and cross.fs), but this would require changing the image format with each optimization we introduce; worse, it would require implementing the optimizations both in the regular Forth compiler (which is also used by gforthmi), and in the cross-compiler, which are not written to share optimizers.

For these reasons the optimizations were eventually all applied below `compile-prim1`. They were set up in a way that allows optionally enabling or disabling most of them, leading to a number of engines with different code generation options applied, and different code generation options enabled by default. Figure 2 may look just as complicated below `compile-prim1` as above it, but actually the lower part is more uniform, because the optimizations are all optional; applying another optimization at that level requires just one copy of the optimization code, if done well.

The only unusual part of code generation that happens at the Forth level is the translation to primitive-centric code (e.g., a reference to a colon definition is replaced with an explicit `call` and the address to be called).

The optimizations in the current development version of Gforth are:

static superinstructions Frequent sequences of primitives are combined into superinstructions [EG03]. E.g., `lit +` → `lit+`.

static stack caching Stack items are kept in registers, with different allocations of stack items to registers at different points in the program [EG04a].

dynamic superinstructions The native code for the primitives in a sequence of threaded code is concatenated together, eliminating many of the NEXTs during execution. Only literal references and control flow use the threaded code [EG03].

native code Similar to dynamic superinstructions, but literal values and branch targets are patched into the native code, and nearly all references to threaded code are eliminated [EG04b].

3 Basic design

This section discusses a number of different inlining implementation strategies.

3.1 Inlining native code

BigForth [Pay91] (and probably other systems) performs inlining by copying the native code of the callee (the called colon definition) into the caller (strategy A in Fig. 3).

Unfortunately, this inlining strategy eliminates most optimization opportunities, because the information used by most optimizations is no longer present in the native code in a readily-used way, and would have to be extracted in complicated ways, if it could be recovered at all; in particular, we cannot use stack caching. BigForth only uses peephole optimization on the code.

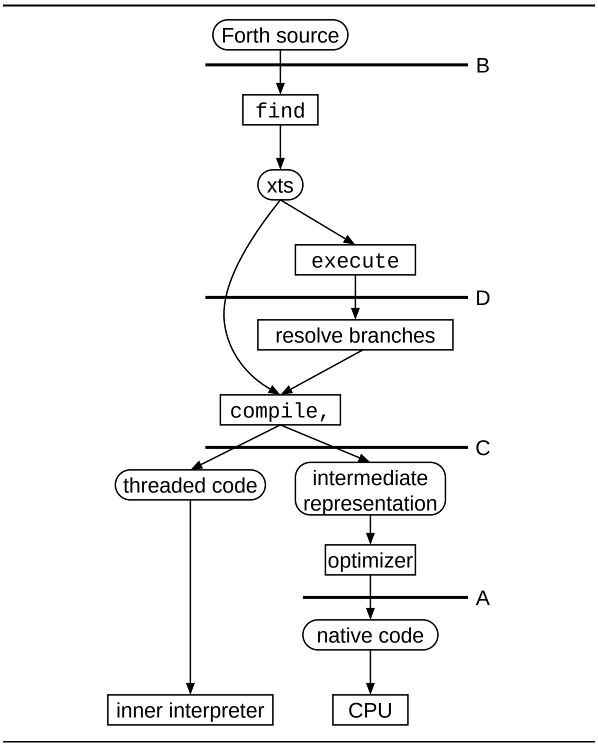


Figure 3: Various inlining options in a simple Forth compiler

A benefit of this method is that it is relatively fast.

One additional complication with inlining native code is that the code can usually not be just copied—it needs to be relocated (e.g., on the 386 architecture calls need their target addresses relocated). There are two ways to deal with this problem: simply don't allow to inline definitions that require relocation (just keep a "no-inline" bit per definition); or set up a relocation table when generating code for the definition, and use that table to perform relocation when inlining (and possibly for other purposes).

3.2 Inlining source code

VFX Forth inlines a definition by running its source text through the text interpreter (strategy B in Fig. 3). The main disadvantage of this method is that the source code is not interpreted in the original context: the search order, the contents of the wordlists, `base` and other variables can be different.

One can partially protect against that by saving the context with the information on the source text, and setting the context to the original context during inlining; however, this is hard to achieve for the contents of the wordlists, and impossible for user-defined context (e.g., user-defined variables that user-defined immediate words may act upon).

The benefit of this approach is that all the inlined code goes through all the compilation, and therefore through all the optimization stages. The drawback of that is, of course, increased compilation time.

3.3 Inlining pre-optimization threaded code

One way to perform inlining is to store the code of each definition in the unoptimized threaded code form, and upon inlining just feed that into the code generation interface (strategy C in Fig. 3), getting all the benefits of the optimizations (which are all performed beneath this level), without suffering the correctness problems and all the compile-time disadvantages of the source-inlining technique.

An additional benefit (over lower-level approaches) is that this technique can be used with both the threaded-code and the native-code engine. Another benefit is that the threaded code already exists in Gforth, so we don't have to develop code and consume memory for storing the code in another intermediate representation.

This is the approach we have taken in our prototype implementation.

One problem with this approach is that the threaded code already contains target addresses for branches and calls, so we have to deal with relocation issues, similar to native code; at least with threaded code the relocation is machine-independent.

3.4 Inlining higher-level intermediate representation

To avoid the relocation issues, one can use a higher-level intermediate code: It should represent the state of compilation after the names have been resolved, and after user-defined immediate words have been executed, to eliminate all context dependences. At the same time, branch resolution should be delayed until after inlining, so that we don't have to deal with relocation issues (strategy D in Fig. 3).

As a concrete example, a word like

```
: abs dup 0< if negate endif ;
```

could have an intermediate representation like

```
: compile-abs
  POSTPONE dup
  POSTPONE 0<
  POSTPONE if
  POSTPONE negate
  POSTPONE endif ;
```

To inline `abs`, one would just have to execute `compile-abs`.

This approach was proposed and partially implemented for RAFTS [EP97, Section 3].

The example above may mislead you into thinking that we can just postpone all words as they come along, but that does not work:

- It would postpone user-defined immediate words, which must be executed at compile time, not later, because they may use (user-defined) context that changes later.
- It would fail to perform the parsing actions of words like s" or [’].

So the way to implement it is to generally hook into compile,, except for some words like if, cs-roll and sliteral, which should compile a word-specific action rather than do their usual thing.

It is not obvious which words need this special treatment. Words that generate primitives with immediate arguments are definitely among them.

In any case, it is not clear that this approach is easier than the threaded-code approach. Moreover, it causes additional complications in the context of Gforth:

We would have the problems with optimizations written in Forth that we mentioned in Section 2.

One may be able to avoid these problems by defining a higher-level code generation interface (and image file format) between Forth and C. But in that case we push even more functionality into the C part, including inlining. Moreover, certain programming practices would not work as usual with such a code generation interface, in particular, creating or modifying control structures without going through the interface words of our higher-level interface (these practices are rare and non-standard, but they still work in traditional-style implementations).

The current image file format can be changed into executable (threaded) code by in-place rewriting. Such a higher-level representation might not have this advantage, and would be more complicated to load.

3.5 What to inline?

There are many possible ways to decide whether a call should be inlined.

The easy way is to leave it to the user: e.g., BigForth allows the user to mark a word as inline, and every call to such a word will then be inlined.

A relatively obvious strategy is to inline only calls to words that occur only once; in this case inlining will also save space, not just time. However, it is not possible to know in Forth whether a word will be referenced again when it is compiled the first time. One way around this problem would be to wait with the actual code generation and inlining decisions until the word or one of its callers is executed; then one would generate code for the executed word, inlining all the words that have been referenced only once until then; later references to the same word will be relatively rare.

An extreme inlining strategy is to inline everything except recursive calls and indirect calls (execute). If we apply that to every word, the native code for the Gforth image grows from 200KB to 7MB; Only generating code for words that are actually executed should reduce the code growth quite a bit, however. Still, this strategy has a bad worst-case behaviour (exponential code growth), and is therefore not usable as a practical inlining strategy, but it is useful to produce best/worst-case numbers for speedup and code growth.

There are lots of other inlining strategies possible. We intend to use our prototype to explore some of them.

4 Details

This section reports some of the approaches we have tried for various problems (including dead-end approaches), and what we learned from them.

4.1 Do we need to inline threaded code?

One approach we considered was to use the threaded code as the basis for inlining, but without generating a threaded-code version of the program with inlining applied; the inlining would only be done on the native code.

One problem with this approach is that it cannot be used, if the native code still accesses the threaded code for literals and for control flow, as happens in the gforth-fast engine.

More precisely, a limited amount of inlining could be performed: The native code would be straight-line, but the IP register would need to follow the control flow in the threaded code (and return addresses would be pushed on the return stack etc.). However, as soon as control flow should happen in the native code (e.g., for a taken conditional branch), this would terminate the inlining; the native code would use the original threaded code for dispatch and execution would continue at the native code corresponding to a non-inlined version of the definition; this restricted approach would provide many of the benefits of inlining (e.g., optimization across calls).

The pure native-code engine (gforth-native) would not have these problems, but even there it is easier to work with the inlined threaded code, for reasons having to do with implementation details of Gforth’s native-code compiler.

4.2 Identifying definitions

In order to be able to inline a colon definition we need to know where it starts and what code belongs to it.

The two contexts used in Gforth exhibit big differences for this problem:

- In the image it is necessary to track control flow in order to notice when a colon definition ends.
- For dynamically-compiled code the call to `finish-code` tells us where a definition ends, and we cannot use control flow earlier, because the targets of (forward) branches are not yet determined when our code sees the branches (as is typical in Forth, they are patched in later).

Overall, the approach we adopted in our inliner is too heavy-weight (also in terms of performance: several passes, unnecessary copying, reanalysis on every inlining), and we may use a more light-weight one in the future.

An alternative approach that can treat the image and dynamically-compiled code in the same way is to determine the code of a definition only during inlining, by just following the control flow at inlining time. The disadvantages of this approach are: You do not know much about the definition beforehand to help inlining decisions (e.g., the size); and you do not know when the end of the definition will come, so some optimizations based on that (like partially inlining the last basic block of the definition) cannot be performed easily; also, analysing the definition each time it is inlined might slow compilation.

Some of these problems could be eliminated by analysing the definition in a separate pass when it is first considered for inlining.

Another uniform approach would be to have end-of-definition markers in the image.

4.3 Internal interface

When we started, the interface to the generator/optimiser of threaded code consisted pretty much of `compile-prim1`. This was fine as long as we were working at the basic block level, rather than the definition level. All we had to do was detect the end of basic blocks.

Detecting the end of colon defintions is much more complicated, not least because we have two completely separate mechanisms for doing it. `Compile-prim1` quickly turned into a moster: It worked in one of two modes, depending on whether we were dealing with image code or code coming from the Forth compiler; most of the functions it called also worked in these two modes.

We solved this problem, by adding a richer interface, which consisted of more than just a function that passes the next primitive. Instead, functions were added to mark branch targets, to mark the end of a basic block, and to mark the end of a colon definition. A layer of code was added between the image relocator (in the loader) and the code generator to keep track of branches and branch targets, and call the end-of-defintion marker when we reached the end of a definition.

Another problem with `compile-prim1` for our new purposes was that there was no separation between building a representation of the code for a definition and generating code for that definition. This was a problem, because when we came across a definition, the main thing we wanted was to get a representation of it that could be optimised or inlined. We also wanted to be able to generate code from this representation without `compile-prim1` thinking it had encountered the definition again. So we broke link between `compile-prim1` and code generation. `Compile-prim1` just builds a data structure representing the definition. If you want to compile that, you need to call the new interface for generating code for a definition. This new interface is, unfortunately, slower, because the data structure for a definition is more elaborate than the simple arrays for a basic block.

4.4 Optimizations

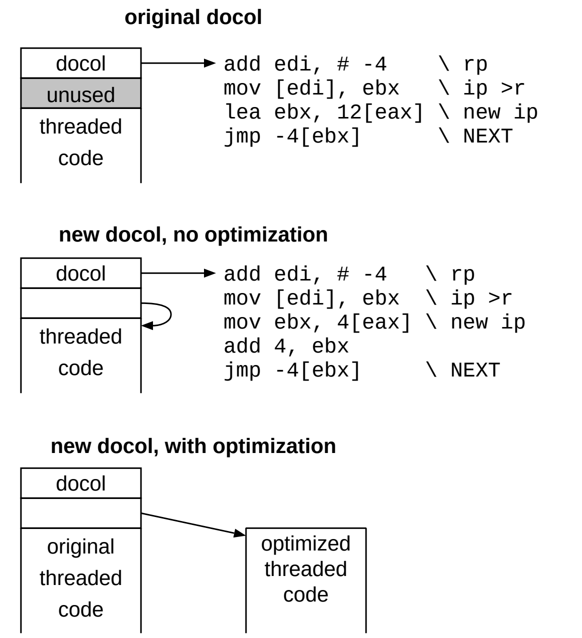
Once the inlining framework was in place, various additional optimizations (beyond inlining) are easy: e.g., tail-call optimization and eliminating unconditional branches around literal data[2], optimizing branches to returns or branches to branches away. Actually, these additional optimizations require much less infrastructure and can therefore be implemented with less compile-time cost; building an optimizer that performs just these optimizations might be more cost-effective than a full-blown inliner.

In some sense, the framework resembles a binary rewriting system such as ATOM [SE94] or Dynamo [BDB00]: It disassembles the threaded code, optimizes it, then reassembles it; in our case the reassembly also results in further code generation steps.

4.5 How to get the new code executed

Once we have generated the optimized threaded code (and, where applicable, the corresponding native code), how do we make sure it is executed?

[2] generated by, e.g., `sliteral`; the literal data stays in the original threaded code and is not copied to the optimized version.



```
add edi, # -4    \ rp
mov [edi], ebx   \ ip >r
lea ebx, 12[eax] \ new ip
jmp -4[ebx]      \ NEXT
```

```
add edi, # -4   \ rp
mov [edi], ebx  \ ip >r
mov ebx, 4[eax] \ new ip
add 4, ebx
jmp -4[ebx]     \ NEXT
```

Figure 4: Finding the optimized threaded code of a colon definition

There are the following issues:

direct control flow (`branch`es, `call`s, etc.) The target addresses in the optimized code must point to targets in the optimized code. The unoptimized version of the same code may be left alone: it should never be executed, but even if it is, execution will eventually reach the optimized code through `execute` and friends (see below). Moreover, it is not always clear that an equivalent optimized version of the target exists (e.g., the return stack contents may differ because of inlining, making all optimized versions of the target non-equivalent).

In spite of these explanations, the current inliner also changes the target addresses in the unoptimized code to point to optimized code.

indirect control flow (`execute` etc.). The xts are still represented as code field addresses (CFAs) of two-cell code fields in the middle of un-optimized code. How do we get `execute` to run the optimized code? There are two cases where `execute` performs threaded code (instead of a primitive, a variable, or somesuch): When the xt of a colon definition is `execute`d, and when the xt of a `does>`-defined word is `execute`d.

Let us look at the `does>` case first: In Gforth the code field of a `does>`-defined word contains the code address `dodoes`, followed by a pointer to the threaded code after the `does>`. We can ensure that the optimized code is used simply by storing a pointer to the optimized code in the second cell (similar to what happens in the *new docol, with optimization* case in Fig. 4).

The colon definition case is handled similarly: We modified the `docol` routine to use a threaded-code pointer in the second cell of the code field; originally this threaded-code pointer points to the body of the word (i.e., the next cell), which contains the original threaded code. The optimizer then changes this pointer to point to the optimized version of the code (see Fig. 4).

returns When executed in the optimized code, the `call` primitive and the `docol` routine push the return address in the optimized code on the return stack, so an unchanged return routine will just return to the right place in the optimized code.

entry points There are two entry points into Gforth's threaded code from the outside: one for the `boot` word, and one for `throw` (for converting OS signals into Forth exceptions). These entry points point to threaded code (not to code fields). Currently our optimizer does not change the entry points, so they still point to unoptimized code, and it is left to indirect or direct control flow to enter the optimized code. By changing the entry points to point to optimized code, we could ensure that unoptimized code is never executed.

5 Preliminary Results

The results in this section were produced on a 2.26GHz Pentium 4 running Linux. We used the inliner with two engines: both use static stack caching, static superinstructions, tail call optimization, and elimination of superfluous unconditional branches[3]; *gforth-fast* uses dynamic superinstructions (native code that still uses the threaded code quite a bit), whereas *gforth-native* produces native code that does not use the threaded code (except for indirect calls). Our current prototype does not inline recursive definitions, or definitions that contain explicit `exit`s.

We varied the amount of inlining performed: if the number of (static) primitives in a colon definition (including the primitives in the definitions it calls) exceeds the inlining limit, it won't be inlined. So, essentially, leaf definitions are inlined, then their callers, etc. until a definition exceeds the inlining size.

[3] An unconditional branch is superfluous, if its target would follow right after the branch in the optimized code; in the original code there is typically some data between the branch and the target, e.g., the string of `S"`.

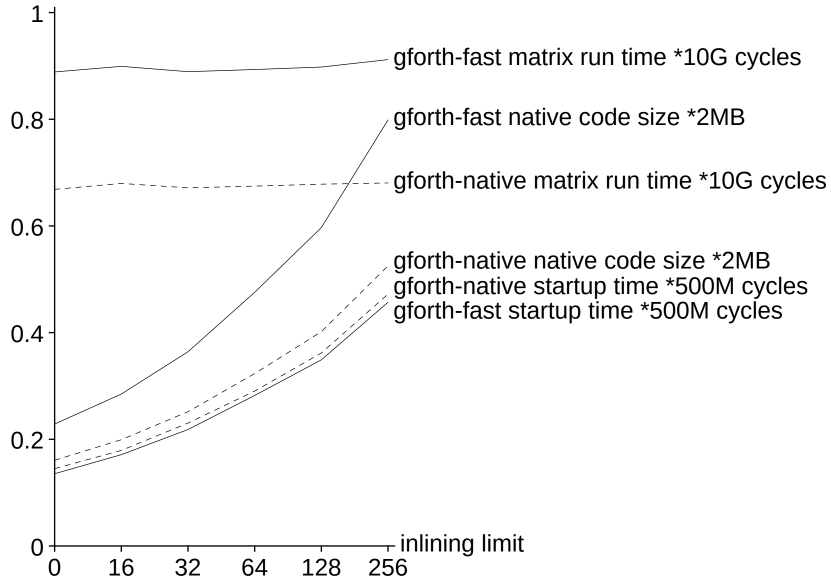


Figure 5: Preliminary results on a 2.26GHz Pentium 4

Note that even with an inlining limit of 0, the inliner produces a copy of the original threaded code, and for both the original and the "optimized" code native code will be generated (so in this case we see twice the native code size that we see without the inliner).

Figure 5 shows run time, startup time, and code size for the two engines, with various inlining limits.

The run-time is for a 500 × 500 integer matrix multiply; we see mostly small slowdowns here, coming from the increased startup time (see below). Apparently inlining (as implemented in our prototype) hardly benefits this benchmark. We have seen some better results in pre-preliminary testing, but do not have any systematic data on them.

The native code size grows with the amount of inlining. Gforth-native enjoys a code size advantage by a factor of 1.42-1.52; it increases with the amount of inlining (the reasons for this increase are not yet clear).

The startup time grows proportionally with the resulting native code size; it is very similar between gforth-fast and gforth-native, probably because they are doing the same thing at the threaded-code level and the native-code part has similar costs. Note that even the startup of gforth-fast with an inlining limit of 256 consumes only 100ms on our benchmark machine, so if we see speedups from inlining, it may well be worth paying that price.

Another interesting result we got was for tail-call optimization: The combination of the image and the prims2x benchmark contains a total of 7774 static returns in the code. Of those, 3727 (48%) were preceeded directly by a call in the optimised code (there may have been an uncoditional branch in the original code that got optimised away), allowing us to apply tail call optimisation. There were an additional 519 calls followed by an unconditional branch (that had not been optimized away before) followed by a return (7%).

6 Related work

Many papers have been written about inlining in the general compiler literature. This section mentions a few of them.

Serrano [Ser97] examines a number of heuristics for determining which calls to inline and presents his own heuristic. He also discusses how to treat recursive functions.

Kaser and Ramakrishnan [KR98] discusses whether the original version should be inlined, or a version where some inlining has already been applied. This only makes a difference for recursive functions, and there the original version should be inlined. They also present some heuristics for inlining which uses a hybrid technique (original-version inlining gets caught in local maxima), and presents some results for code growth and inlining effectiveness.

De Bosschere et al. [BDGK94] deal with the problem of (optimizing away) entry actions (type checking, unboxing) in languages with run-time type checking. One of the optimizations proposed is partially inlining the entry actions into the callers (where they can be optimized away), but the authors report that this leads to significant code bloat. Goubault [Gou94] also attacks the problem of unboxing and boxing with partial inlining.

7 Conclusion

Calls and returns are the most frequently executed control flow words in Forth code, and are therefore the main rason for the short basic block length in Forth code. In order to increase the effectiveness of various optimizations, the basic block length has to be increased, e.g., through inlining.

At what level should inlining be performed? Our prototype inliner applies inlining at the before-optimization threaded-code level, and we think that this is the right decision.

What experiences did we gain when we wrote our prototype? Even for our native-code compiler, it was a good idea to perform the inlining on the threaded code first. Identifying (the end of) definitions proved to be a problem. We had to expand the simple code generation interface to convey more control flow information. The framework we developed for inlining could also be used for other control flow optimizations. And we identified how to get the new code executed, in particular, how to deal with the docol routine.

Our preliminary timings are disappointing, but we hope to achieve better results in the future.

References

[Alm86] Thomas Almy. Compiling Forth for performance. *Journal of Forth Application and Research*, 4(3):379–388, 1986.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[BDGK94] Koen De Bosschere, Saumya Debray, David Gudeman, and Smapath Kannan. Call forwarding: A simple interprocedural optimization technique for dynamically typed languages. In *Principles of Programming Languages (POPL '94)*, pages 409–420, 1994.

[EG03] M. Anton Ertl and David Gregg. Implementation issues for superinstructions in Gforth. In *EuroForth 2003 Conference Proceedings*, 2003.

[EG04a] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *Interpreters, Virtual Machines and Emulators (IVME '04)*, pages 7–14, 2004.

[EG04b] M. Anton Ertl and David Gregg. Retargeting JIT compilers by using C-compiler generated executable code. In *Parallel Architecture and Compilation Techniques (PACT' 04)*, pages 41–50, 2004.

[EP96] M. Anton Ertl and Christian Pirker. RAFTS for basic blocks: A progress report on Forth native code compilation. In *EuroForth '96 Conference Proceedings*, St. Petersburg, Russia, 1996.

[EP97] M. Anton Ertl and Christian Pirker. The structure of a Forth native code compiler. In *EuroForth '97 Conference Proceedings*, pages 107–116, Oxford, 1997.

[Gou94] Jean Goubault. Generalized boxings, congruences and partial inlining. In *Static Analysis Symposium (SAS '94)*, volume 864 of *LNCS*, pages 147–161. Springer, 1994.

[Koo89] Philip J. Koopman, Jr. *Stack Computers*. Ellis Horwood Limited, 1989.

[KR98] Owen Kaser and C. R. Ramakrishnan. Evaluating inlining techniques. *Computer Languages*, 24:55–72, 1998.

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

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI '94)*, pages 196–205, June 1994.

[Ser97] Manuel Serrano. Inline expansion: *When* and *How*. In *Programming Languages, Implementation and Logic Programming (PLILP)*, volume 1292 of *LNCS*, pages 143–157. Springer, 1997.