How to Implement Words (Efficiently)

M. Anton Ertl[*]
TU Wien

Abstract

The implementation of Forth words has to satisfy the following requirements: 1) A word must be represented by a single cell (for `execute`). 2) A word may represent a combination of code and data (for, e.g., `does>`). In addition, on some hardware, keeping executed native code and (written) data close together results in slowness and therefore should be avoided; moreover, failing to pair up calls with returns results in (slow) branch mispredictions. The present work describes how various Forth systems over the decades have satisfied the requirements, and how many systems run into performance pitfalls in various situations. This paper also discusses how to avoid this slowness, including in native-code systems.

1 Introduction

We all know how to implement words efficiently, as demonstrated by our Forth system implementations. Right?

When measuring various Forth systems for another work [EP24, Figure 11], I found that SwiftForth 4.0.0-RC87 was surprisingly slow for some benchmarks, in particular CD16sim (written by Brad Eckert, part of the appbench benchmark suite[1]). Eventually I found the reason for the slowness of CD16sim, and reported the problem and its cause to Forth, Inc. They swiftly released SwiftForth 4.0.0-RC89, which fixed the CD16sim slowness and also produced significant speedups for several other application benchmarks[2] (see Fig. 1).

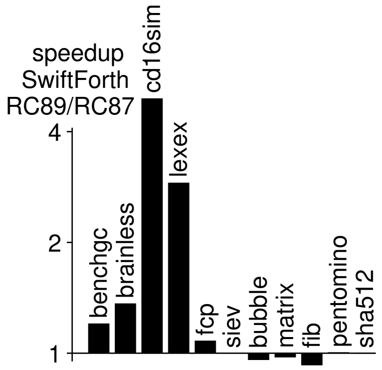

Figure 1: Speedup of SwiftForth 4.0.0-RC89 over SwiftForth 4.0.0-RC87 on a TigerLake CPU

While the fix performed in 4.0.0-RC89 is enough to make CD16sim perform as I expect from the small benchmarks, there are still cases where various Forth systems (including SwiftForth) experience performance pitfalls. These problems have to do with the way words are implemented in these Forth systems. So in this paper I look at various ways to implement words, and how they are affected by the performance pitfalls.

Section 2 discusses some of the performance pitfalls of modern processors. Section 3 discusses requirements of Forth words that have led system implementors to fall into performance pitfalls. Section 4 discusses the implementation techniques of indirect-threaded code, which is the base of the design of many modern systems. Section 5 takes a look at the variety of implementation techniques in modern systems. Section 6 shows performance results on a number of microbenchmarks, and discusses how these results stem from the performance pitfalls. Finally, Section 7 discusses related work.

2 Performance pitfalls

There are various reasons why acceleration mechanisms do not work every time. In the present work I have encountered the following reasons, and, as we can see, in many cases these reasons can be avoided.

2.1 False sharing between I and D-cache

Caches do not cache each byte individually, but larger units called cache lines, typically 64 bytes long. This has advantages, such as reducing hardware overhead and increasing the effectiveness of

---

[*] `anton@mips.complang.tuwien.ac.at`

[1] http://www.complang.tuwien.ac.at/forth/appbench.zip

[2] Interestingly, the changes doe not speed up the 6 other benchmarks I have used recently (siev, bubble, matrix, fib, pentomino, and sha512); the source code for these 6 benchmarks is smaller and less typical of idiomatic Forth source code. This is a reminder that we should also look at application benchmarks for evaluating the performance of a Forth system.

the cache for spatial locality, but also a disadvantage: false sharing [SB93]. If two pieces of data are in the same cache line, but are accessed through different coherent caches, and at least one of these pieces of data is written to, a phenomenon known as false sharing happens:

The write to the line in cache A will invalidate the cache line in cache B through the cache-coherence protocol. When the access (even just a read) to the cache line in cache B happens, it will fetch the modified line from cache A through the cache-coherence protocol, but depending on the protocol it may take some (expensive) broadcasting to discover where the up-to-date contents of the cache line is, so this is expensive.

This mechanism is designed for communicating data between cores, i.e., one core writes some data and the other reads it (true sharing). When the data accessed in the two caches is actually non-overlapping, and just happens to be in the same cache line by accident, this is known as false sharing.

Normally false sharing is something that plagues programmers of multi-threaded programs. But in Forth we have been plagued by false sharing between the I-cache and the D-cache on architectures that have coherent I-caches (these days, IA-32, AMD64, and s390x), ever since separate I and D-caches were introduced with the Pentium in 1993. That is because many Forth systems place code close to written data. As we will see, it is possible to avoid that.

Many systems have taken measures to eliminate the common reasons for executed code being close to written data, but in the absence of complete separation the problem rears its head in various not so common cases, as we will see.

The cost of one cache ping-pong between I and D-cache (i.e. one cycle of executing and storing) seems to be on the order of 400 cycles on recent Intel P-cores.

2.2 Return misprediction

Modern processors predict branches, and if the prediction is correct, the branch is executed in 0–1 cycles. One of the branch predictors used is the (hardware) return-address stack[3] [KE91]: a call pushes the return address on the return-address stack, and the return instruction predicts that it will branch to the address it has from the hardware return stack. However, this prediction is later verified when the return instruction actually sees the real return address (coming from (cached) memory indexed through `%rsp` in case of the AMD64 `ret` instruction).

The return-address stack predicts very well if every call is paired with a return to the predicted address.

However, if the return address pushed by a call is pulled and used for something else, and the next return should return to the return address pushed by an earlier call, the return will mispredict, as will all the returns to even earlier calls. So pulling one return address can lead to multiple mispredictions. Likewise for the push-return technique for performing indirect branches.

Using a return address for something other than returning is a venerable Forth implementation technique, as we will see, but on systems that use hardware call and return for colon definitions, they lead to slowness ever since hardware return-address stacks were introduced in the 1990s.

Another venerable Forth implementation technique is to change the return address for skipping over some data or code (e.g., in implementations of `sliteral`); this results in one misprediction when returning to the changed return address with the return instruction, but at least the remaining hardware return-address stack will still predict correctly.

The cost of a branch misprediction is on the order of tens of cycles.

3 Requirements

Forth has certain requirements for the implementation of words. One is that some words do not just have an execution semantics (i.e., code), but in a number of words that execution semantics refers to data that can be written to: the words defined with `create` (without and with `does>`), `variable`, `2variable`, `fvariable`, `buffer:`, and `defer`. Words defined with, e.g., `field:` may also deal with data (depending on the implementation) in addition to code, but that data is read-only, and therefore should at least not lead to false sharing problems.

Both the code and the data of a word are represented in a single cell, the execution token (xt) of a word. In particular, `execute` needs to jump to the code and that code needs to access the data.

The xt is also used for `compile,`. One might use the same mechanism for performing `compile,`d code as for `execute`, and in indirect-threaded code that is done, but one can also make `compile,` more intelligent and let it generate better code. This means that compiled code may suffer less from pitfalls than `execute`d code.

The xt is also used for `defer`red words; it's possible to use an optimizing mechanism here, but it's not clear that the `defer`red word is performed often enough relative to the number of `is`/`defer!` changes to justify an optimizing mechanism. And

---

[3] This is a microarchitectural mechanism that should not be confused with the Forth return stack.

```
: bd bdoes> @ ;
: cd cdoes> @ ;
create x 0 ,
<builds y 0 , bd
create z 0 , cd
```

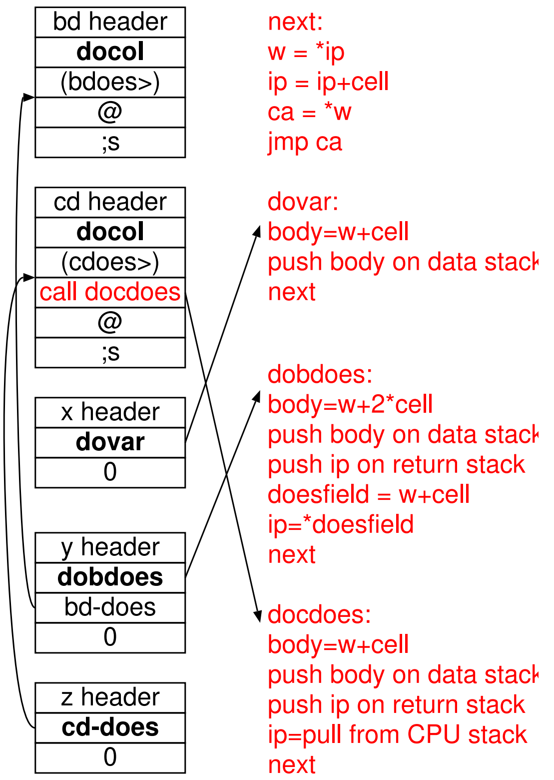

Figure 2: Implementation of words with associated data in indirect-threaded code. Code field in **bold**, native (pseudo-)code in red.

if we implement words, xts, and `execute` to avoid performance pitfalls, a straightforward implementation of `defer`red words will also avoid these pitfalls.

4 Indirect-threaded code

This section explains how the requirements are met in Forth systems that use indirect-threaded code. The techniques used by several modern systems are based on those used for indirect-threaded code.

Figure 2 shows the source code and implementation of three words `x`, `y`, and `z` and also some of the defining words used for defining them. In indirect-threaded code all execution, whether with `execute` or running `compile,`d code, performs an indirect jump to the address in the **code field** for every word; the native code that is jumped to in this way determines the behaviour of the word, so we have **docol** for colon definitions, **dovar** for words that push the body address (variables and `create`d words), **docon** for constants, etc.

`X` is a created word (without `does>`), so it has dovar in the code field, which pushes the body address of `x`. How does dovar achieve this? The dispatch code of the previous word sets a register (called *W* in the Forth literature) to point to the code field. This happens on every path that jumps to dovar, whether it is `execute`, dodefer, or, in compiled code, the *next* routine at the end of the previous word (*next* is shown in Fig. 2). Dovar then computes the body address from *w* and pushes it on the data stack. Other doers (e.g., docon) also use *w* to get access to the data, or, in the case of docol, to the threaded code.

4.1 `Does>`

Words with `does>`, such as `y` and `z`, require access to the threaded code after the `does>` (the *doescode*) in addition to access to the body and the native-code doer. There have been two solutions used in indirect-threaded code systems; this paper uses the names `bdoes>` and `cdoes>` (and related names) to make it clear which solution is meant.

The first one (used for `y`) reserves an additional cell (the *doesfield*) right after the code field. The doesfield points to the doescode. `Y`'s doer *dobdoes* uses *w* to compute the body address (which starts two cells after the code field for `y`) and to load the address of the threaded code after the `bdoes>` from the doesfield. `Y` is defined with `<builds`, which allocates the additional cell for the doesfield. `Bdoes>` is intended to be used with `<builds`, and you cannot use it with `create` and get the usual results. Fig-Forth provides `<builds` and a `does>` that is equivalent to `bdoes>`.

The disadvantage of the `<builds`...`bdoes>` solution is the extra cell necessary for every word defined with `<builds`. So Dean Sanderson [Moo80, page 72] and Mike LaManna[4] came up with the alternative mechanism, shown here for `z`: Instead of having an extra cell, let the code field of `z` point right after the `cdoes>`; of course, there must still be native code there, and we have to get to the doer, so the usual approach is to put a native-code call to the doer *docdoes* right after the `(cdoes>)`, and let that call be followed by the threaded code for the Forth code after the `cdoes>`. *Docdoes* pulls the return address of the call, and since call is right before the doescode, *docdoes* now has the doescode. As we will see, this call-pull technique is still widespread and is a major cause of false sharing and return mispredictions.

The way that doescode is determined is the main difference between docdoes and dobdoes.

---

[4] Thanks to Leon Wagner for reporting this contributor.

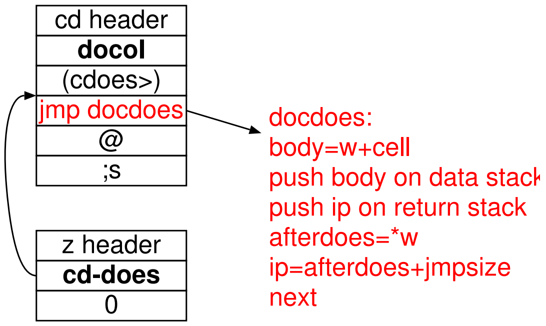

Figure 3: An implementation variant for `cdoes>` that uses a jmp instead of a call.

Note that in threaded code, there are no call-return pairs around this usage of call-pull, so you do not see mispredicted returns from this usage. And the machines for which this technique was invented had no caches, and therefore no false sharing.

This approach works with `create`, so no additional `<builds` is needed, and it was therefore eliminated. This technique was introduced in the short time between fig-Forth and Forth-79 and apparently took the Forth world by storm. Forth-79 already standardized `create`...`does>`.

5 Alternative implementation techniques

5.1 Avoiding return mispredictions

Instead of having a call right after the `cdoes>`, one can have a jump. Then recovering the address of the code after the `does>` is not possible with a pull. However, you can determine the address from *w* (see Fig. 3).

5.2 Direct-threaded code (ITC style)

The same techniques used for `cdoes>` can also be used for the code field in order to implement direct-threaded code: Have a jump or call at the code field that jumps to the doer, and then get the body address either from *w* or with the call-pull technique.

This approach (using jumps) has been used for direct-threaded code in Gforth up to Gforth 0.5 [Ert93]. These versions of Gforth use direct-threaded code on selected architectures and indirect-threaded code on all others.

For primitives, the threaded code points directly to the native code of the primitive, not to a jump or call. The advantage of this direct-threaded code over indirect-threaded code is that there is one load less in *next*; this benefit works for primitives, while for other words the load is replaced by a jump or call.

This approach puts a piece of native code just in front of the body of every word, and if the body is written to, this results in false sharing between I-cache and D-cache. Therefore Gforth switched to indirect-threaded code for architectures with coherent I-cache (in particular, IA-32); after Gforth 0.5 it switched to hybrid direct/indirect threading [Ert02], which combines the benefits of both approaches.

5.3 Subroutine-threaded code

Many native-code systems conceptually are optimized subroutine-threaded code systems [For20, Section 5.1.1], and the way words are implemented are often based on subroutine-threaded code.

In subroutine-threaded code a primitive is invoked through a native-code call, both for compiled code and for `execute`. For words with data, these systems use the same approach as direct-threaded code: a call to the doer just before the data. If the data is written, this results in a round of cache ping-pong.

Another problem with this approach is that the call-pull pattern for getting the body address hurts in a subroutine-threaded system, because such a system actually uses return instructions that are then mispredicted.

Both problems do not just occur with words defined with `does>`, but, like in direct-threaded code, with all words with a doer and data (false sharing only results in a slowdown on modern CPUs if the data is written to).

SwiftForth and VFX Forth use this approach, but they often avoid calling the words with data in the body, and therefore both performance problems. However, in some cases they fail to avoid these problems. The CD16Sim problem of SwiftForth 4.0.0-RC87 was one case where the problem was not avoided, and it was fixed in RC89 by avoiding it.

Could not at least the call-pull problem be avoided in the same way as for direct-threaded code? Unlike in direct-threaded code, no *w* register is set when running compiled subroutine threaded code. A workaround that works for both `execute`d and compiled code would be quite complex, and given that there are other options (see below), to my knowledge nobody has used such an approach.

5.4 Avoid body

One of the ways in which subroutine-threaded and native-code systems reduce the problems is by reducing the number of words where you need a doer and data.

In particular, colon definitions are just called directly instead of through a doer.

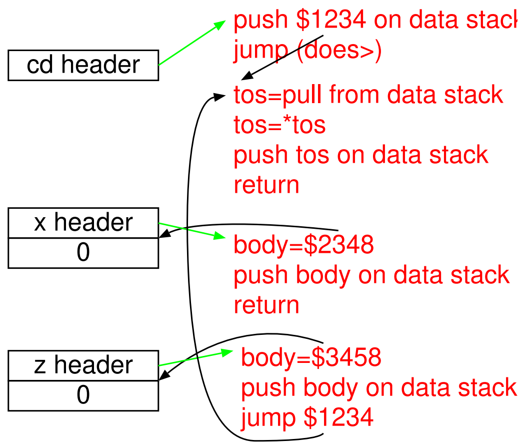

Figure 4: Trampolines for `x` and `z`. While the header points to the trampoline, this pointer is not followed at run-time (so it is not a code field), but at text-interpretation time. The code is shown as pushing and popping, but usually this works with registers

For words where the data does not change, in particular, constants and field words, it is relatively straightforward to generate native code for the behaviour of the word (including the data). E.g., a constant `c` with the value 5 could be defined in a way that results in the same code as

```
: c 5 ;
```

5.5 Trampolines

For the remaining words, instead of having just a call or jump to the doer before the body of the word and then needing some way to recover the body address, we can provide the body address as a literal and then jump to the doer. This technique is called a trampoline in gcc, and is used there for the same purpose: to represent a tuple of code and data with just one address.

Once the body address is provided as a literal, there is actually no need to put the trampoline right in front of the data. Instead, it can be put anywhere, e.g., in a separate code section, or otherwise away from frequently-written data (see Fig. 4).

This approach solves both the false-sharing problem and the return-misprediction problem. This is a recommended approach. It is used by ntf/lxf (by Peter Fälth) and by FlashForth[5].

5.6 Intelligent `compile,`

In traditional indirect-threaded code, `compile,` always performs `,`, and in a simple subroutine-threaded system, it compiles a call to the word.

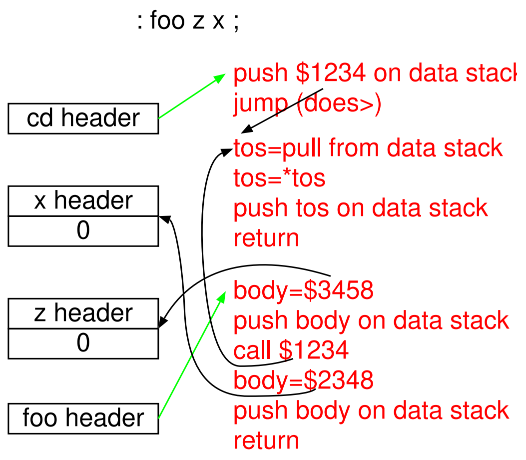

Figure 5: Code compiled for `foo` with an intelligent `compile,`.

An intelligent `compile,` generates code specialized for the word type or possibly even the individual word [Ert02, PE19]. In the present discussion, an intelligent `compile,` can compile `x` as the literal that pushes the body address of `x`, and `z` as the literal that pushes the body address followed by a call to the doescode (not to `z`), see Fig. 5.

This means that in compiled code uses of `x` and `z` result neither in false sharing nor in return mispredictions. SwiftForth uses this approach for `does>`-defined words since SwiftForth 4.0.0-RC89 and it solves the CD16sim slowdown that earlier versions suffered from.

With `compile,` implementations for dovar and `does>`-defined words as suggested, the trampolines for our examples can be generated by producing the same code as:

```
:noname x ; \ trampoline for x
:noname z ; \ trampoline for z
```

In case you are wondering whether the trampoline is needed for this code generation: It is not: `X` and `z` are only `compile,`d, not `executed` in this code. Tail-call optimization is needed to turn the call to the doescode for `z` into a jump to the doescode.

One useful property of the intelligent `compile,` is that it allows to use completely different mechanisms for `compile,` and `execute`. E.g., since version 0.6 Gforth uses primitive-centric direct-threaded code (plus a long list of optimizations based on that) for `compile,`d code, but uses indirect-threaded dispatch for `execute` and `deferred` words [Ert02].

If the different implementations of `execute` and `compile,` lead to different dispatch mechanisms, the trampoline-generating approach outlined above

---

[5] `news:<c2588b8c811fd3ae75d3976c3a927fc3@www.novabbs.com>`

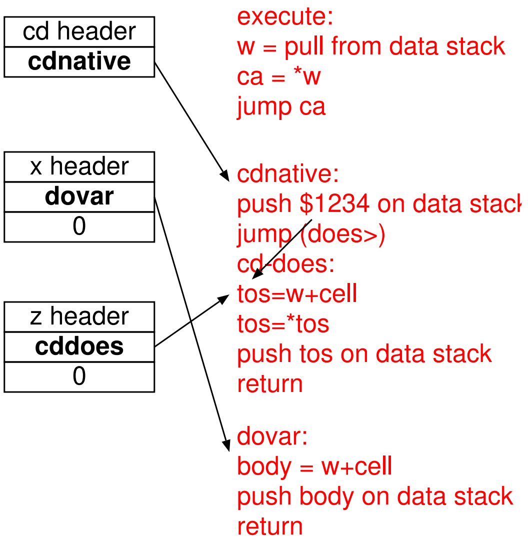

Figure 6: A native-code system with a code field containing a code address (as in ITC)

does not work or needs to change. But ideally you design the mechanism for `execute` such that trampolines are unnecessary (see Section 5.8)

However, the difference between the mechanisms also means that just because we don't see performance problems in compiled code, does not mean that they don't appear in `execute`d code. We will see examples in Section 6. In particular, SwiftForth's `compile,` avoids the performance problems in compiled code in RC89, but such problems still are present when `execute`ing words.

5.7 `Deferred` words

A straightforward way to implement `defer`red is with a simple one-cell body that contains the xt, and that xt is invoked with the same kind of dispatch as `execute`. This results in all the performance pitfalls of the `execute` implementation on that system, but one can build a system without such performance pitfalls, e.g., with trampolines, so this is the recommended approach.

Another approach is to implement a deferred word in a native-code system as a jump to the current target of the `defer`red word. This means that `is` (and `defer!`) change the code, resulting in true sharing between the data and instruction cache, which causes slowdowns on all architectures, and cannot be eliminating by separating code and data. Lxf-1.6 uses this approach.

5.8 Native-code address field

Fforth[6], which is in its infancy, is going to be a native-code system that uses a code field that contains the code address for use with `execute` and for `defer`red words. The dispatch of `execute` and for calling `defer`red words first sets w to the code field address (CFA), then loads the contents of the CFA (the code address), and jumps to the code address. The doer then can determine the body from the contents of w, like in indirect-threaded code. Since Fforth is a native-code system there is no difference between a system-defined doer and the doescode; the doescode starts with computing the body from w, and making the body the top-of-stack, then continues with the native code for the Forth code after the `does>`.

For compiled code, Fforth uses an intelligent `compile,`. A simple way to call a word is to load the CFA of the `compile,`d word into w and then call the doer, but I expect that in most cases faster implementations will be used. See Fig. 6.

This approach can avoid all the usual performance pitfalls of native-code systems, just like the trampoline, but costs only one data cell per word, whereas the trampoline approach typically consumes more memory and is a little more work to implement.

5.9 Always have a doesfield

Memory is no longer as tight as when `create`...`does>` was introduced at the end of the 1970s, so Gforth has had two cells between the header of a word and its body from the get-go in 1992; in indirect-threaded code engines before the new header [PE19], the first cell is used for the code field and the second cell is used for the doesfield [Ert93], always allowing to use `bdoes>` for such engines, rather than the `cdoes>` variants used with direct-threaded code engines.

With the new header, there are again two cells in the neck: the code field, and the hm field (header methods, which we previously called vt [PE19]). Hm points to a method table that contains the doesfield as one of its fields. This means that dodoes performs one more indirection for getting to the doescode than with the old header. However, in the usual case (compiled code) the extra indirection is resolved at compile time, so it does not cost in that case.

5.10 Double-indirect threaded code

Returning to threaded-code systems, another way to deal with the need in `does>`-defined words for

---

[6] `https://github.com/AntonErtl/fforth`

doer, body, and doescode without needing a doesfield is to repeat the benefit of the indirection in indirect-threaded code by introducing another indirection [Ert02]. The xt in *w* is close to the body, *w* `@` (stored in *w2*) is close to the doescode, and *w2* `@` points to dodoes, which is then performed and accesses the body through *w* and the doescode through *w2*.

This approach would cost an additional indirection over indirect-threaded code on every `execute` or `defer`red word, but the idea was that this would not happen for compiled code, because that would use direct-threaded code [Ert02]. We did not go with this approach in Gforth, and instead stayed with always having a doesfield. To my knowledge, nobody has implemented this approach.

6 Measurements

This section presents some microbenchmarks and reports how different systems perform. As always, microbenchmarks are not intended to represent application performance, but to shine a spotlight on certain performance characteristics.

The measurements were done on a Xeon E-2388G (Rocket Lake); I measured similar results on a Golden Cove and a Tiger Lake (all three are Intel P-cores). The Forth systems measured are gforth-fast 0.7.9_20240817 (gforth), iforth 5.1-mini (iforth), lxf 1.6-982-823 (lxf-1.6), SwiftForth 4.0.0-RC89 (sf RC89), SwiftForth 4.0.0-RC87 (sf RC87) and VFX Forth 64 5.43 (vfx). When both SwiftForth versions produced similar results, only one of them is shown, under the name sf.

Shortly before EuroForth, I also received lxf 1.7-172-983 from Peter Fälth, and I repeated the measurements of `defer`red words with that, and list the results of the new version as lxf-1.7.

The columns shown are the cycles, instructions, I-cache load misses, D-cache load misses, and branch mispredictions performed per iteration of the microbenchmark.

Here are the Forth words that the microbenchmarks measure:

```
create x 0 ,

: d1 ( "name" -- )
 create 0 ,
does> ( -- addr )
;

d1 z1

: d2 ( "name" -- )
  create 0e f,
 does> ( -- )
  1e dup f@ f+ f! ;

d2 z2

0 constant my0

defer w ' my0 is w
```

For each of the words `x`, `z1` and `z2` there is a microbenchmark that compiles it and one that `execute`s it. Moreover, for `w` we have two comp/exec pairs of microbenchmarks: One that changes what `w` performs once per invocation of `w`; and one that keeps that word always the same.

6.1 The original problem

```
: bench-z1-comp ( -- )
  iterations 0 ?do
    1 z1 +!
  loop ;
```

		cache misses		branch	
cycles	inst.	I	D	mispred	system
8.2	34.0	0.0	0.0	0.0	gforth
9.0	6.6	0.0	0.0	0.0	iforth
6.4	15.0	0.0	0.0	0.0	lxf-1.6
6.5	14.0	0.0	0.0	0.0	sf RC89
434.2	15.0	2.0	2.0	1.0	sf RC87
7.7	4.6	0.0	0.0	0.0	vfx

This is the microbenchmark inspired by CD16sim. SwiftForth RC87 suffers from false sharing and mispredicted returns, and RC89 fixed that problem.

6.2 ... and it's execute variant

```
: bench-z1-exec ( -- )
  ['] z1 iterations 0 ?do
    1 over execute +!
  loop
  drop ;
```

		cache misses		branch	
cycles	inst.	I	D	mispred	system
9.4	41.0	0.0	0.0	0.0	gforth
16.5	49.6	0.0	0.0	0.0	iforth
7.0	17.0	0.0	0.0	0.0	lxf-1.6
431.1	24.0	2.0	2.0	1.0	sf
449.8	17.6	2.0	2.0	1.0	vfx

When `execute`ing `z1`, both sf and vfx suffer from false sharing and return mispredictions thanks to using the call-pull technique.

6.3 Is VFX always fine on compiled code?

```
: bench-z2-comp ( -- )
    iterations 0 ?do
        z2
    loop ;
```

		cache misses		branch	
cycles	inst.	I	D	mispred	system
15.4	42.0	0.0	0.0	0.0	gforth
11.4	9.6	0.0	0.0	0.0	iforth
12.1	17.0	0.0	0.0	0.0	lxf-1.6
12.6	17.0	0.0	0.0	0.0	sf RC89
248.8	22.0	2.0	1.0	1.0	sf RC87
231.6	15.6	1.0	1.0	1.0	vfx

One might expect that `z2` has the same performance pitfalls as `z1`, and that's roughly true for the SwiftForth variants. However, VFX manages to avoid the performance pitfalls for `z1` with inlining, but in the `z2` case the FP code apparently disables inlining in VFX, it calls the call in the header of z2, and therefore suffers from the usual slowdowns of the call-pull technique.

6.4 What about iForth?

```
: bench-z2-exec ( -- )
    ['] z2 iterations 0 ?do
        dup execute
    loop ;
```

		cache misses		branch	
cycles	inst.	I	D	mispred	system
10.4	49.0	0.0	0.0	0.0	gforth
449.5	49.6	2.0	2.1	0.0	iforth
13.5	19.0	0.0	0.0	0.0	lxf-1.6
428.3	26.0	2.0	2.0	1.0	sf RC89
249.5	30.0	2.0	1.0	1.0	sf RC87
228.2	16.6	1.0	1.0	1.0	vfx

Looking at the code, iforth seems to use the call-pull technique, too, and therefore suffers from false sharing; it does not suffer from return mispredictions, because it does not use `ret` for implementing Forth's `exit` and `;`.

It's unclear why the two sf versions produce such differences in the number of cycles; a wild guess is that the actual slowdown depends on the exact placement of the word within the cache line. In any case, neither result is good, and we should try to avoid even the smaller slowdown.

6.5 Compiled `created` words are fast

```
: bench-x-comp ( -- )
    iterations 0 ?do
        1 x +!
    loop ;
```

		cache misses		branch	
cycles	inst.	I	D	mispred	system
6.9	11.0	0.0	0.0	0.0	gforth
8.6	6.6	0.0	0.0	0.0	iforth
7.8	5.0	0.0	0.0	0.0	lxf-1.6
1.4	3.0	0.0	0.0	0.0	sf
7.7	4.6	0.0	0.0	0.0	vfx

None of the systems exhibit a big performance problem for a compiled `created` word, but the performance of iforth, lxf-1.6, and vfx may still merit an investigation.

6.6 … but once you `execute` …

```
: bench-x-exec ( -- )
    ['] x iterations 0 ?do
        1 over execute +!
    loop drop ;
```

		cache misses		branch	
cycles	inst.	I	D	mispred	system
7.0	28.0	0.0	0.0	0.0	gforth
16.5	49.6	0.0	0.0	0.0	iforth
6.0	17.0	0.0	0.0	0.0	lxf-1.6
442.8	24.0	2.0	2.0	1.0	sf
221.1	17.6	1.0	1.0	1.0	vfx

Both sf and vfx run into false sharing here, as well as a return misprediction.

6.7 What about `defer` and `is`?

```
: bench-w-comp ( -- )
    ['] my0 ['] drop iterations 0 ?do
        w over is w
    loop
    2drop ;
```

		cache misses		branch	
cycles	inst.	I	D	mispred	system
7.0	22.5	0.0	0.0	0.0	gforth
9.2	19.6	0.0	0.0	0.0	iforth
427.0	21.5	2.0	1.0	0.3	lxf-1.6
6.7	10.5	0.0	0.0	0.0	lxf-1.7
435.9	19.5	2.7	2.0	1.0	sf
205.3	11.1	1.0	1.0	0.5	vfx

In this benchmark sf and vfx suffer from false sharing and return misprediction resulting from the call-pull technique.

Lxf-1.6 suffers from true sharing due to writing to the jump that is then executed. CPUs also don't have as good branch prediction mechanisms for code that patches jumps as they have for indirect branches, so the patching results in a significant increase in branch mispredictions compared to, e.g., Gforth, which uses an indirect jump in `dodefer` and `lit-perform` (the primitive used by the `compile,` implementation of `defer`red words).

Lxf-1.7 uses the indirect jump approach, and therefore does not suffer from the performance pitfalls of lxf-1.6.

6.8 … in combination with `execute`

```
: bench-w-exec ( -- )
    [’] w dup [’] my0 [’] drop
    iterations 0 ?do
        3 pick execute over is w
    loop
    2drop drop ;
```

		cache misses		branch	
cycles	inst.	I	D	mispred	system
6.9	28.5	0.0	0.0	0.0	gforth
16.4	40.6	0.0	0.0	0.0	iforth
429.0	22.5	2.0	1.0	0.3	lxf-1.6
11.1	15.5	0.0	0.0	0.0	lxf-1.7
445.2	28.5	2.5	2.0	1.0	sf
228.9	21.1	1.0	1.0	1.5	vfx

The results in this case are very similar to the bench-w-comp case, but vfx suffers from an additional return misprediction: it’s `execute` implemention uses push-ret instead of an indirect branch to branch to its target.

6.9 What about `defer` without `is`?

```
: bench-w-nois-comp ( -- )
    iterations 0 ?do
        w drop
    loop ;
’ z1 is w bench-w-nois-comp
```

		cache misses		branch	
cycles	inst.	I	D	mispred	system
8.4	35.0	0.0	0.0	0.0	gforth
15.5	42.6	0.0	0.0	0.0	iforth
5.4	12.0	0.0	0.0	0.0	lxf-1.6
5.0	12.0	0.0	0.0	0.0	lxf-1.7
29.4	16.0	0.0	0.0	1.0	sf
27.2	11.6	0.0	0.0	1.0	vfx

In this microbenchmark no data is written, so there is no cache-consistency traffic from false or true sharing. This allows us to see the undiluted penalty of the return mispredictions resulting from call-pull in SwiftForth and VFX.

This is the best case for the lxf-1.6 defer implementation (patching jump), but the fact that the more mainstream lxf-1.7 defer implementation is just as fast (actually slightly faster) even in this case means that the cost of cache consistency traffic from the jump-patching implementation cannot be compensated, even if `is` is used rarely.

6.10 … in combination with `execute`

```
: bench-w-nois-exec ( xt -- )
    iterations 0 ?do
        dup execute drop
    loop
    drop ;
’ z1 is w ’ w bench-w-nois-exec
```

		cache misses		branch	
cycles	inst.	I	D	mispred	system
8.4	41.0	0.0	0.0	0.0	gforth
25.5	62.6	0.0	0.0	0.0	iforth
6.0	13.0	0.0	0.0	0.0	lxf-1.6
10.0	17.0	0.0	0.0	0.0	lxf-1.7
32.2	24.0	0.0	0.0	1.0	sf
65.9	21.6	0.0	0.0	2.0	vfx

With `execute`, vfx suffers from an additional misprediction per iteration, which is reflected in the cycle count.

Lxf-1.7 takes 4 instructions more and consumes 4 cycles more per iteration than lxf-1.6 for this microbenchmark. I looked at the resulting code, and communicated some improvement suggestions[7] to Peter Fälth; he then produced three implementation variants for `defer`red words that perform this benchmark in 13–14 instructions and 7 cycles, and two of them perform as well or better than lxf-1.7 on the other defer-based microbenchmarks. This demonstrates that the disadvantage of a defer implementation that uses indirect jumps can be made very small in the cases where the deferred word is `execute`d or called through another deferrred word, too. The code for implementing these variants consisted of a few lines each.

7 Related work

While indirect-threaded code has been used in Forth by 1971 at the latest, the canonical papers on direct-threaded code [Bel73] and indirect-threaded code [Dew75] came only later.

Kogge [Kog82] describes the path from subroutine-threaded code to indirect-threaded code (and the benefits of these steps in the memory-constrained systems of the time).

The Forth mainstream went the other direction and went to direct-threaded code [Ert02] and dynamic superinstructions (a kind of native code) with stack caching [EG04] in Gforth, or for native-code compilers in iForth, lxf, SwifthForth, and VFX Forth. The reasons are that with increasing RAM size the pressure to minimize program memory became smaller; moreover, with increasing cell size the

---

[7] Generate specialized code for the deferred word rather than using a trampoline to a generic dodefer, and eliminate a tail call while doing that.

size advantage of threaded code dwindled or even became a size disadvantage.

While there are several works describing the header structure and execution mechanisms of early Forth systems [Moo74, Kog82, Tin13b, Zec84, Tin13a, Tin17], most widely-used systems since the 1990s except Gforth [Ert93, Ert02, PE19] have seen relatively little material published about the parts that correspond to the inner interpreter in a threaded-code system. Faulkner has sketched a generator that allows exploring a variety of implementation options [Fau23].

Scott and Bolosky [SB93] quantified the cost of false sharing. Kaeli and Emma [KE91] proposed the return-address stack for predicting return targets, which appeared in actual hardware a few years later.

8 Conclusion

For subroutine-threaded and native-code compilers, the trampoline approach avoids problems with cache consistency and return mispredictions. An alternative is to use a code field even in a subroutine-threaded or native-code system.

Either approach is best combined with an intelligent `compile,` for efficient compiled code.

`Defer`red words should be implemented with an indirect jump (or call) rather than a direct jump that is patched by `is`.

References

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973. 7

[Dew75] Robert B.K. Dewar. Indirect threaded code. *Communications of the ACM*, 18(6):330–331, June 1975. 7

[EG04] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *Interpreters, Virtual Machines and Emulators (IVME '04)*, pages 7–14, 2004. 7

[EP24] M. Anton Ertl and Bernd Paysan. The Performance Effects of Virtual-Machine Instruction Pointer Updates. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:26, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. 1

[Ert93] M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Lázně (Marienbad), 1993. 5.2, 5.9, 7

[Ert02] M. Anton Ertl. Threaded code variations and optimizations (extended version). In *Forth-Tagung 2002*, Garmisch-Partenkirchen, 2002. 5.2, 5.6, 5.10, 7

[Fau23] Glyn Faulkner. 4g and FAIL. In *39th EuroForth Conference*, 2023. 7

[For20] Forth, Inc. *SwiftForth Reference Manual*, 2020. 5.3

[KE91] David R. Kaeli and Philip G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *The 18th Annual International Symposium on Computer Architecture (ISCA)*, pages 34–42, Toronto, 1991. 2.2, 7

[Kog82] Peter M. Kogge. An architectural trail to threaded-code systems. *Computer*, pages 22–32, March 1982. 7

[Moo74] Charles H. Moore. Forth: A new way to program a mini-computer. *Astron. Astrophys. Suppl.*, 15:497–511, 1974. 7

[Moo80] Charles H. Moore. FORTH, the last ten years and the next two weeks. *Forth Dimensions*, I(6):60–75, March/April 1980. 4.1

[PE19] Bernd Paysan and M. Anton Ertl. The new Gforth header. In *35th EuroForth Conference*, pages 5–20, 2019. 5.6, 5.9, 7

[SB93] M. Scott and W. Bolosky. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, volume 57, page 41, 1993. 2.1, 7

[Tin13a] C.H. Ting. *Inside F83*. Offete Enterprises, fourth edition, 2013. 7

[Tin13b] C.H. Ting. *Systems Guide to figForth*. Offete Enterprises, third edition, 2013. 7

[Tin17] C.H. Ting. *Footsteps in an Empty Valley*. Offete Enterprises, fourth edition, 2017. 7

[Zec84] Ronald Zech. *Die Programmiersprache FORTH*. Franzis, München, first edition, 1984. In German. 7