Interpreter vs. Compiler Performance at Run-Time

M. Anton Ertl
anton@mips.complang.tuwien.ac.at
TU Wien

Abstract

Both “interpreter” and “compiler” cover a wide range of implementation techniques, and in some cases one can argue where to draw the line. Consequently, both approaches also have a wide performance spread, leading to comparable performance in some cases. The present work looks closely at the SwiftForth JIT compiler and the Gforth interpreter. Even with dynamic superinstructions and IP-update optimizations, Gforth suffers from interpretive overhead, but that is compensated by Gforth’s additional sophistication in caching stack items in registers, an optimization that SwiftForth does not employ.

1 Introduction

The following properties are desired for programming language implementations:

- Execution speed
- Compilation speed
- Cheap development and maintainence (of the programming language implementation)
- Portability/retargetability

Programming language implementation techniques have been traditionally been classified as interpreters or compilers.[1] Both classes encompass a wide range of implementation techniques.

The conventional wisdom is that compilers are not just faster than interpreters, but that the two classes are so far apart that the claim

> There is a bigger performance gap between a performance oblivious interpreter and a high-performance interpreter than between a high-performance interpreter and a JIT.

in a draft version of a paper led a reviewer to ask for some support for this claim. In a later draft this claim had morphed into its final version [7]:

> While [interpreters] cannot compete in execution performance with JIT compilers or ahead-of-time compilers, a fast interpreter is not that far away: e.g., with the IP update optimizations of the present work, Gforth has similar performance to the SwiftForth JIT compiler and to gcc -O0.

which led one reviewer to speculate “It might be that SwiftForth [...] is simply a very weak compiler.” and the statement “None of these can let us conclude that, in general, interpreters are not lagging far behind compilers.” Another reviewer was very impressed by the results (also shown in Section 2).

These reactions show a widespread belief that interpreter performance and compiler run-time performance are islands separated by a vast ocean.

[1] There is also metacompilation, where an interpreter is partially evaluated [9]; this approach has become practical in recent years thanks to huge efforts by developers of systems like GraalVM/Truffle [12]. However, in the present work we focus on more traditional approaches, and leave metacompilation to further work.

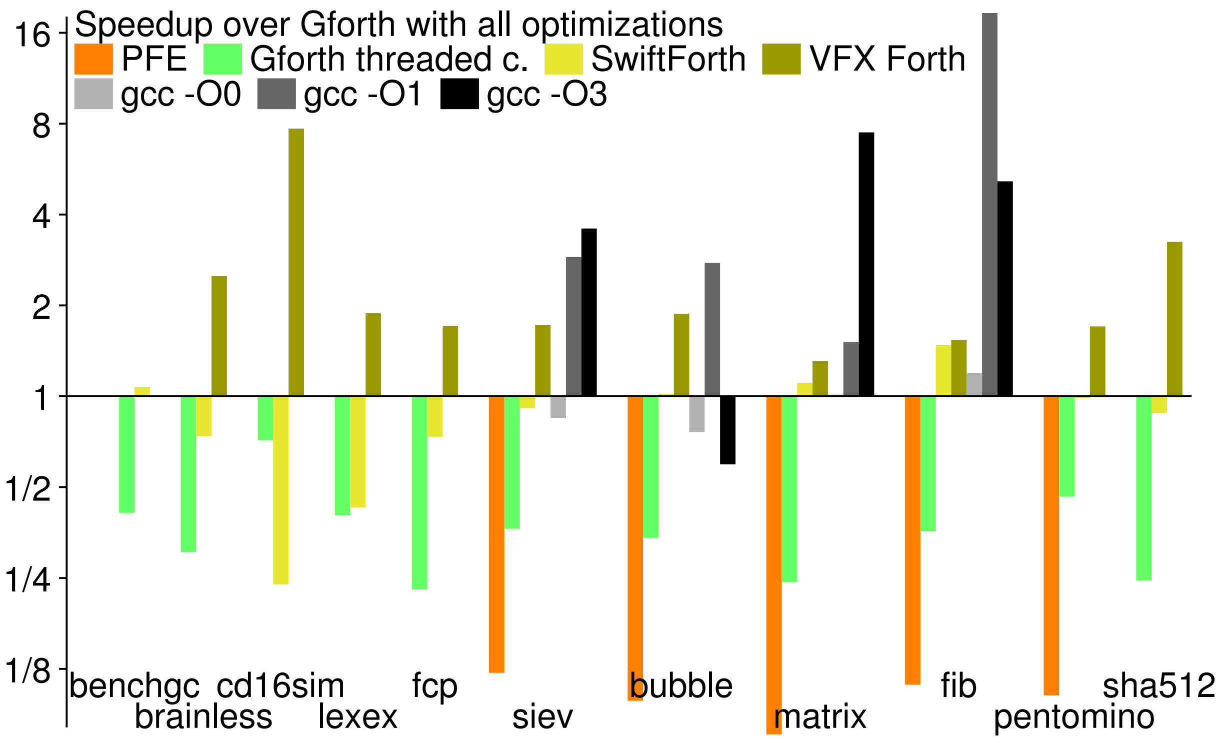

Figure 1: Speedup (above the baseline) or slowdown (below the baseline) of several Forth systems and gcc over the fastest Gforth version, on Tiger Lake (Core i5-1135G7). If a benchmark does not work on a system, no bar is shown for the combination.

I have written the present paper to address this belief.

First I look at some performance results (Section 2), then take a closer look at two of the involved systems (Section 3).

2 Performance

Figure 1 shows the same data as Figure 10 of our upcoming paper [7], but uses a different baseline system (in the speedup numbers the run time of the baseline is divided by the run time of the system specified for the bar): here the baseline is the fastest Gforth version, the one that includes the optimizations from our recent paper [7]. We consider it to be a virtual-machine (VM) interpreter, because it needs to access the representation of the VM code at run-time in order to access immediate operands and to perform control flow. However, as we will see there is a degree of machine-code generation involved.

The systems and compilers compared with the baseline are:

PFE is an interpreted Forth system written in C that uses one C function per VM instruction implementation. PFE is designed to rely on explicit register allocation (a GCC extension) for performance, but unfortunately, for AMD64 no explicit register definitions have been added yet. We use PFE-0.33.71.

Gforth threaded c. This is the baseline Gforth with the option `-no-dynamic`, which means that it falls back to using direct-threaded code [1]; this option also disables stack caching. This is similar in implementation and performance to Gforth around the year 2000.

SwiftForth, VFX Forth Two commercial Forth systems with JIT compilers. We measured SwiftForth x64-Linux 4.0.0-RC87 and VFX Forth 64 5.43.

gcc-12 Various optimization options for GCC 12.2. Manually written C code for four of the benchmarks is available and was used for generating these results. For gcc the results do not include the compile time (unlike for the Forth systems).

Coming back to the claims that prompted this paper, while PFE is not performance-oblivious, the baseline Gforth has a higher speedup over it for the benchmarks where PFE works than VFX Forth (the fastest JIT compiler we measured) has over the baseline.

VFX Forth performs extensive inlining and allocates data stack items to registers within basic blocks. This leads to a speedup over Gforth by about a factor of 2 in most benchmarks. Idiomatic Forth code consists of short definitions, resulting in many calls, so inlining is particularly effective. *cd16sim* contains a large number of (implicit) calls to an empty definition, making the inlining of VFX particularly effective here.

SwiftForth, another JIT compiler, is typically in the same ballpark as the fastest Gforth; for *bubble* and *pentomino* the performance is so similar that the SwiftForth bar is barely visible. We will discuss SwiftForth in depth in Section 3.

`gcc -O0` produces performance similar to the baseline. `gcc -O1` shows a good speedup. `gcc -O3` is better on some benchmarks, but worse on others. We have looked at the two slowdown cases. For *bubble*, `gcc -O3` auto-vectorizes, and the result is that there is partial overlap between a store and a following load, which results in the hardware taking a slow path rather than performing one of its store-to-load forwarding optimizations. For *fib*, we have not found the reason for the slowdown.

3 A tale of two Forth systems

This section provides a closer look at SwiftForth and Gforth and the tradeoffs in their creation, and how this affects performance.

3.1 SwiftForth

SwiftForth is a commercial Forth system from Forth, Inc. Forth, Inc. (the first commercial Forth vendor) developed a number of interpreters using indirect-threaded code [2] from the 1970s until the 1990s [11]. These systems were written in Forth with a foundation of (Forth) assembly language.

With the introduction of SwiftForth during the 1990s Forth, Inc. switched from threaded-code interpreters to native-code compilation (aka JIT compilation). The intention of this change was improved performance, and that also shows in SwiftForth's name. And it delivers: In most benchmarks, SwiftForth is 2–4 times faster than Gforth threaded code (which probably has performance similar to the threaded-code polyForth II system that preceded SwiftForth). Fortunately, while SwiftForth's source code is proprietary, it is delivered with SwiftForth, which makes it easier to study the compilation techniques used.

The basic compiler of SwiftForth concatenates the native code of the primitive Forth words in a definition into the native code for that definition. The primitives are still written in assembly language, and are probably very similar to the code for the primitives in polyForth II, with one exception: In polyForth II each primitive ends with a threaded-code dispatch, while in SwiftForth it ends with a native-code `ret` instruction (which is not copied in the concatenation).

In addition, for control-flow primitives the native code is first copied and the the target offsets are patched into the copied code; similarly, for literal numbers the number is patched into the copied code for the primitive `(literal)`. This is the fundamental difference to Gforth's interpreter-based approach, which does not patch native code, but instead keeps the interpreted code around and accesses it when control flow or literal values are needed.

In addition, SwiftForth optimizes pairs of Forth primitives to better code with optimization rules like

```
OPTIMIZE DUP +         SUBSTITUTE 2*
```

The result of the substitution can be subject to another optimization rule, resulting in the optimization of longer sequences. There are 346 optimization rules used in the version of SwiftForth we measured; in many cases this requires to also define the substitution word and/or a word that performs additional compile-time work when an optimization rule triggers. These optimizations include tail-call optimization. In the version of SwiftForth we measured, files containing a total of 1819 lines contain most of the optimizations.

So while SwiftForth's compiler is not the most sophisticated one in existence (and simplicity is a major goal in SwiftForth development), it is far from being "very weak".

Forth native-code compilers have suffered from running into microarchitectural pitfalls due to legacy techniques for implementing certain features of Forth. In particular, the measured version of SwiftForth implements Forth's `does>` in a way that puts native code close to written-to data, resulting in ping-ponging between the I-cache and the D-cache due to false sharing. It also pops the return address of a call instead of returning to it, resulting in branch mispredictions in subsequent returns. These pitfalls are responsible for the low performance of SwiftForth on at least *cd16sim*.

These pitfalls have been reported to Forth, Inc., which fixed them (for the common case) in SwiftForth-4.0.0-RC89, but that version appeared too late for producing new results and working them into this paper.

This episode demonstrates that on modern CPUs it's not enough to reduce the number of executed instructions, you also have to be aware of microarchitectural pitfalls and avoid them. The *bubble* result of `gcc -O3` demonstrates that even a project that puts massive manpower into optimization can run afoul of such problems.

One cost of SwiftForth's assembly-based approach is that each of the i386 and the x64 ports has about 7000 lines of architecture-specific files.

3.2 Gforth

Gforth is a non-commercial free software project and was developed mainly on Unix systems starting in 1992. Among the goals of Gforth was that it should be efficient and available on many machines. These goals were initially achieved by implementing Gforth mostly in Forth, with an indirect-threaded code interpreter, i.e., along traditional Forth implementation techniques, but using GNU C instead of assembly language for writing the interpreter foundation.

Given the wide range of general-purpose computer architectures of the 1990s, our portability goals made machine-specific code without a fallback to machine-independent code unattractive, so we did not switch to native-code compilers.

From 2001 to 2005 we implemented a number of optimizations and enabling changes: switching to primitive-centric threaded code [3], static superinstructions [8], dynamic superinstructions and stack caching [4, 6]. This frenzy was followed by a long period of consolidation and focussing on other topics, but eventually additional optimizations were added: generalized constant folding [10] and the IP-update optimizations [7]. These improvements were research-driven, i.e., they used Gforth as a research vehicle, but they then became production features.

A static superinstruction combines a sequence of primitives into a better primitive, like simple substitution rules for SwiftForth's optimizer. The version of Gforth that we have measured uses 55 static superinstructions; the reasons why we don't use more are: One of the benefits of static superinstructions is subsumed by stack caching, so we use static superinstructions only in cases where we expect an additional benefit; our implementation of static superinstructions does not work as well with stack caching as we would like; adding more static superinstructions increases the compile time of Gforth.

Dynamic superinstructions optimize a sequence of primitives by concatenating their native code, but without the threaded-code dispatch. The effect on straight-line code is similar to the effect of SwiftForth's basic compiler. One difference is that immediate operands of primitives (e.g., of literals) and the targets of control flow are not patched into the native code, but are still accessed through the virtual-machine instruction pointer (IP), which points to the same place as in the original threaded code. Performing the next primitive in sequence is achieved by falling through to the code of the next primitive, but all other control flow is performed through a threaded-code dispatch. A benefit of this approach is that one can fall back to plain threaded code for a single primitive (e.g., because it is not relocatable) by appending the threaded-code dispatch code to the dynamic superinstruction preceding it.

As implemented until 2023, every primitive still updated the IP. Since 2023, we have implemented a number of IP-update optimizations, which produce big speedups (up to a factor of 3) on loop-dominated benchmarks on modern hardware.

Forth is a stack-based language. Stack caching represents the stack in several different ways, each with a different number of stack items in registers. This allows to implement variants of primitives that perform fewer memory accesses and fewer stack pointer updates than the base variant of the primitive that uses the same representation on entry and on exit. Gforth uses up to three registers for stack items on AMD64. This optimization has a part of the effect of VFX's register allocation for data-stack items in a basic block.

Generalized constant folding allows to perform optimizations such as turning division by a constant into a multiplication by the reciprocal, but in the interest of brevity I will not discuss it further here.

Performancewise, compared to SwiftForth, Gforth suffers from having to perform control flow through threaded-code dispatch and having to access immediate operands in memory through IP, but it benefits from stack caching which SwiftForth does not have.

The main benefit of Gforth's approach is portability: E.g., Gforth ran out of the box when AMD64, ARM A64, and RISC-V became available to us (in 2003, 2016, and 2017, respectively), and with a little work (typically less than an hour) all performance features could be activated. By contrast, SwiftForth's AMD64 port only started in 2020 and it is still in Beta testing, and the only other port is the IA-32 port. However, SwiftX, the cross-compiler for embedded systems, is available for 11 cores from 8 base architectures.

Gforth's approach comes at a cost in complexity, though. A rough estimate based on the sizes of various files is that about 5000 lines of code are spent on static superinstructions, dynamic superinstructions, stack caching, and IP-update optimizations. The IP-update optimizations alone have inserted 864 lines and deleted 316 lines [7].

One development interesting in the present context is that at one point we worked on completely eliminating the IP from Gforth [5], turning it into what was later called a copy-and-patch compiler [13]. However, that approach appeared to be too brittle (no fallback option that would work under all circumstances), so we did not turn it into a production feature.

3.3 Example

To make things more concrete, here's a piece of Forth source code that demonstrates the differences between SwiftForth's and Gforth's code generation. Consider the Forth definition:

```
: squared dup * ;
```

This definition defines the word `squared` to perform the primitives `dup` and `*`; then the definition ends (and at run-time the execution returns). The value to be squared is passed on the data stack, and `dup` pushes another copy of that value on the data stack. `*` then pops these two copies, multiplies them, and pushes the result.

For this example neither SwiftForth nor Gforth have static superinstructions, so we can look at the native code for the individual primitives (or, with stack caching, primitive variants):

Src	SwiftForth	Gforth
dup	lea -8(%rbp),%rbp mov %rbx,0(%rbp)	mov %r8,%r15
*	mov 0(%rbp),%rax mul %rbx mov %rax,%rbx lea 8(%rbp),%rbp	imul %r15,%r8
;	ret	mov (%r14),%rbx add $8,%r14 mov (%rbx),%rax jmp *%rax

The big picture is that stack caching leads to much shorter code for `dup` and `*` for Gforth, but Gforth performs more instructions for the return from `squared` to its caller; i.e., we see the interpretive overhead in this return.

For SwiftForth the details are: the data-stack pointer is in `%rbp`, and the `lea` instructions update the data-stack pointer. The top of the data stack is in `%rbx`. `mul` multiples `%rax` with the argument, and puts the result in `%rax`.

For Gforth, with one data stack item in a register (representation 1), the top-of-stack is in `%r8`; with two data stack items in registers (representation 2), the second item is in `%r8` and the top item is in `%r15`. In this example, `dup` starts out in representation 1, and finishes in representation 2; `*` starts in representation 2 and it finihes in representation 1.

For the return, the return-stack[2] pointer is in `%r14`, and IP is in `%rbx`. So the return first loads the IP from the return stack, updates the return-stack pointer, and then performs a threaded-code dispatch.

4 Conclusion

The Gforth interpreter shows performance in the same ballpark as the SwiftForth compiler. When looking at the details, we see that Gforth, despite using several optimizations that reduce the interpretive overhead, still suffers from the remainder of this overhead; in particular, the overhead on control flow and when dealing with immediate operands. However, apparently stack caching

[2] Forth stores return addresses on a separate stack so that they are not in the way of accessing data on the data stack.

(implemented in Gforth, but not in SwiftForth) provides enough speedup to compensate for the interpretive overhead slowdown.

In addition, with either approach one should avoid falling prey to microarchitectural pitfalls.

References

[1] James R. Bell. “Threaded Code”. In: 16.6 (1973), pp. 370–372.

[2] Robert B.K. Dewar. “Indirect Threaded Code”. In: 18.6 (June 1975), pp. 330–331.

[3] M. Anton Ertl. “Threaded Code Variations and Optimizations (Extended Version)”. In: *Forth-Tagung 2002*. Garmisch-Partenkirchen, 2002. URL: `https://www.complang.tuwien.ac.at/papers/ertl02.ps.gz`.

[4] M. Anton Ertl and David Gregg. “Combining Stack Caching with Dynamic Superinstructions”. In: *Interpreters, Virtual Machines and Emulators (IVME ’04)*. 2004, pp. 7–14. URL: `https://www.complang.tuwien.ac.at/papers/ertl%26gregg04ivme.ps.gz`.

[5] M. Anton Ertl and David Gregg. “Retargeting JIT compilers by using C-compiler generated executable code”. In: *Parallel Architecture and Compilation Techniques (PACT’ 04)*. 2004, pp. 41–50. URL: `https://www.complang.tuwien.ac.at/papers/ertl%26gregg04pact.ps.gz`.

[6] M. Anton Ertl and David Gregg. “Stack Caching in Forth”. In: *21st EuroForth Conference*. Ed. by M. Anton Ertl. 2005, pp. 6–15. URL: `https://www.complang.tuwien.ac.at/papers/ertl%26gregg05.ps.gz`.

[7] M. Anton Ertl and Bernd Paysan. “The Performance Effects of Virtual-Machine Instruction Pointer Updates”. In: *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Ed. by Jonathan Aldrich and Guido Salvaneschi. Vol. 313. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 14:1–14:26. ISBN: 978-3-95977-341-6. URL: `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.14`.

[8] M. Anton Ertl et al. “vmgen — A Generator of Efficient Virtual Machine Interpreters”. In: *Software—Practice and Experience* 32.3 (2002), pp. 265–294. URL: `https://www.complang.tuwien.ac.at/papers/ertl+02.ps.gz`.

[9] Octave Larose et al. “AST vs. Bytecode: Interpreters in the Age of Meta-Compilation”. In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). URL: `https://doi.org/10.1145/3622808`.

[10] Bernd Paysan. “Constant Folding für Gforth”. In: *Vierte Dimension* 35.2 (2019), p. 17. URL: `https://wiki.forth-ev.de/lib/exe/fetch.php/vd-archiv:4d2019-02.pdf`.

[11] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. “The Evolution of Forth”. In: *History of Programming Languages (HOPL-II) Preprints*. SIGPLAN Notices 28(3). 1993, pp. 177–199.

[12] Thomas Würthinger et al. “Practical partial evaluation for high-performance dynamic language runtimes”. In: *SIGPLAN Not.* 52.6 (June 2017), pp. 662–676. ISSN: 0362-1340. URL: `https://doi.org/10.1145/3140587.3062381`.

[13] Haoran Xu and Fredrik Kjolstad. “Copy-and-Patch Compilation”. In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021), 136:1–136:30. URL: `https://fredrikbk.com/publications/copy-and-patch.pdf`.