A Look at Gforth Performance

M. Anton Ertl*
TU Wien

Abstract

Gforth used to be an traditional threaded-code system. In the last decade we integrated a number of performance features into Gforth. Several of them were evaluated individually, but an evaluation with a more global perspective has been missing until now. This paper fills this void: We have measured the performance of Gforth releases from 0.5.0 to 0.7.0, on a wide variety of machines, and employing a wide variety of GCC versions for compiling Gforth. We present that data and give explanations for the performance differences.

1 Introduction

Up until and including gforth-0.5.0, Gforth employed quite traditional implementation techniques: Indirect threaded code or, on some architectures, direct threaded code.

Then we added a number of performance-improving techniques, which were released with Gforth 0.6 and Gforth 0.7: Primitive-centric hybrid direct/indirect threaded code [Ert02] was mainly an enabler for further optimizations. Dynamic superinstructions with replication [RS96, PR98, EG03b, EG03a] probably have the most significant effect on performance; these were all present in Gforth 0.6. Static superinstructions were added in Gforth 0.6.2, and static stack caching [EG04, EG05] in Gforth 0.7.0.

Moreover, Gforth-0.7.0 includes a number of changes to make these and other optimizations (in particular, explicit register allocation) more effective: Automatic build tuning, workarounds for GCC bugs, and some architecture-specific improvements.

In this paper, we take an overall look at these changes and their performance effects on various architectures.

Unfortunately, during the same time GCC was also "optimized", and that often resulted in significantly lower performance for Gforth. We found workarounds for some of these problems, but the question remains how effective they are across GCC versions and architectures. So in this paper we also look at how Gforth performs when compiled with various GCC versions on various architectures.

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

2 Setup

2.1 Gforths

We compare four versions of Gforth, with an additional three variants produced by running these versions with an option that turns off a new feature. The Gforth versions and variants we looked at were:

0.5.0 Uses traditional indirect or direct-threaded code. Direct-threaded code is only supported on some architectures, indirect threaded code on all of them.

0.6.1 no dynamic This variant uses primitive-centric hybrid direct/indirect threaded threaded code. It's still threaded code, but now colon definitions are compiled into a `call` primitive followed by an address, variables are compiled to `lit` followed by the address, etc. I.e., all threaded-code pointers point to primitives. Dynamic superinstructions with replication are disabled in this version (by running Gforth with `--no-dynamic`) in order to make it as close in performance to 0.5.0 as is easily possible, and to allow isolating the effect of that optimization.

0.6.1 This variant enables dynamic superinstructions with replication [RS96, PR98, EG03b, EG03a] on platforms where they are available. This feature works as follows: for a sequence of code without branches, the native code of the primitives is copied to a new place, and these native code fragments are concatenated. The direct threaded code points to these copies of the native code, not the originals. Most of the NEXTs are left away. Only when there is a branch, `call` or `execute` in the threaded code, a NEXT is needed. This feature reduces the number of NEXTs executed and increases the indirect branch prediction accuracy of the remaining NEXTs.

Architecture	CPU	Clock rate	
Alpha	21264B	800MHz	8MB L2
AMD64	Opteron 270	2000MHz	1MB L2, like Athlon 64 X2
	Xeon 5450	3000MHz	2 × 6MB L2, like Core 2 Quad
ARM	Xscale IOP 80321	600MHz	
IA32	Pentium 4 (Northwood)	2267MHz	512KB L2
	Athlon MP	2000MHz	512KB L2, like Athlon XP
	Opteron 270	2000MHz	1MB L2, like Athlon 64 X2
	Xeon 5450	3000MHz	2 × 6MB L2, like Core 2 Quad
IA64	Itanium II	900MHz	
PPC	PPC7447A (G4)	1066MHz	512KB L2
	PPC970 (G5)	2000MHz	
PPC64	PPC970 (G5)	2000MHz	

Figure 1: Machines

0.6.2 no superinst This variant has the same performance features as 0.6.1. Static superinstructions, the new performance feature of 0.6.2, are disabled.

0.6.2 This version adds static superinstructions, a platform-independent feature. Static superinstructions essentially combine a sequence of primitives into one primitive. Unlike dynamic superinstructions, which are created at Gforth run-time, static superinstructions are created beforehand and built into the Gforth engine. Gforth 0.6.2 uses 27 and 0.7.0 uses 13 static superinstructions.

0.7.0 simple stack caching This version tests if the explicit register allocation option works, and uses it if it works. Explicit register allocation tells GCC what registers to use for various VM registers (stack pointers etc.). Otherwise GCC often allocates the VM registers in memory, so explicit register allocation can provide a significant speedup on some architectures. Gforth 0.7.0 also contains several other performance improvements that are often somewhat specialized: E.g., it supports indirect branch target alignment for dynamically generated code, providing a speedup on Alpha; there are also performance improvements in mixed-precision division. And a number of architectures have better support in 0.7.0, allowing them to employ dynamic superinstructions.

0.7.0 This variant adds multi-state static stack caching: instead of keeping the number of stack items in registers the same (usually one item in the top-of-stack register) all of the time, the number of stack items in registers can vary to minimize the number of loads from and stores to the stack memory, as well as stack pointer updates. Most architectures have too few registers available in a way usable with GCC and therefore can use only at most one register. On the PPC and PPC64 architectures we use up to three registers.

All versions of Gforth were compiled without enabling non-default performance features (such as explicit register allocation on versions before Gforth 0.7.0). That is the way that Linux distributors compile Gforth (and most Linux users get Gforth through their distribution rather than building it themselves). On the other hand, most Windows users probably use the binary package built by Bernd Paysan, and that uses non-default build options (in particular `--enable-force-reg` for explicit register allocation) to improve performance. So, the presented results are not representative for typical Windows installations.

A few other features that are not related to performance and are not used for the benchmarks (e.g., the C library interface) were disabled in order to help make the resulting binaries portable. We compiled the four Gforth variants once for each architecture and GCC version, and then ran the resulting binaries on all machines of that architecture.

2.2 Hardware and OS

Figure 1 shows the hardware we used. Several machines were able to run binaries for two architectures. All of these machines were running under various versions of Linux, on various versions of the Debian distribution. All machines had enough RAM to run the benchmarks without swapping.

2.3 Benchmarks

Figure 2 shows the benchmarks we use. These are all application benchmarks of significant size, and

Program	Author	Description
bench-gc 1.0	Anton Ertl	Garbage Collector
brainless 0.0.2	David Kuehling	Chess
cd16sim v11	Brad Eckert	CPU emulator
fcp 1.31-64	Ian Osgood	Chess
lexex	Gerry Jackson	Scanner Generator

Figure 2: Benchmark programs used

hopefully their usage patterns are more representative of other CPU-intensive applications than some of the smaller benchmarks that are often used (and that have quite different behaviour from these and other application benchmarks).

Each benchmark was run three times (on each combination of Gforth variant, GCC version, and machine), and the median of the three results was used further on.

In a few graphs we show results for individual benchmarks, but in most graphs we show an aggregate of all benchmarks. We use the geometric mean for aggregation (with each benchmark having the same weight) [FW86].

Brainless produces different results on 32-bit and 64-bit systems, and probably would produce different run-times even on a system that was always equally fast in 32-bit and 64-bit mode. Therefore we did not include brainless in the aggregate if we compare 32-bit and 64-bit systems.

2.4 GCC versions

We tried to compile Gforth with as many GCC versions as possible. Fortunately, there is a wide variety of GCC versions available on Debian, and they can be installed simultaneously. In addition, there were some manually installed GCCs available on some architectures.

2.5 Graphs

All graphs are scaled such that the highest-performing system gets speed 1. Also, all graphs are scaled logarithmically.

For graphs where each data point represents a Gforth variant with no reference to a specific compiler, the fastest-performing variant out of those that ran is shown. This should show what the various versions of Gforth are capable of when not hindered by GCC performance bugs.

In some graphs data points are missing, either because building that version of Gforth did not work, or because one of the benchmarks failed (for all of the Gforth compilations under consideration).

If a missing data point lies between two others in a line graph, the line is drawn from the point before to the point after, which is incorrect: It suggests that the performance of the missing point is in the middle, but actually there was no performance at all for that point; however, trying to make these cases more visible would probably add more confusion than it would help, so we decided against it.

If a missing point is at the start or the end of the line, it is just not shown. In some cases, there is only one point in the line, which is then not shown. Instead you see the label of the "line" to the right of where the point is.

3 Results and Analysis

3.1 Overall performance

Figure 3 shows a performance summary: Each line represents an architecture/machine combination. The points on each line show the performance of different Gforth versions/variants, for each the fastest `gforth-fast` binary that the different compiler versions produced.

Overall, we can see that Gforth performance has improved significantly between 0.5.0 and 0.7.0, e.g., by a factor of more than 3 for IA32 Xeon 5450, and that factor seems pretty typical.

Another overall observation we can make is that we managed to build all Gforth versions on all machines, even on architectures that were not available to us for testing when we released the old versions of Gforth (like ARM or PPC64), or that were not even released when Gforth 0.5.0 was released in 2000, like IA64 (released in 2001) and AMD64 (2003). This shows that Gforth achieves its goal of portability very well.

3.2 Gforth versions

Looking closer, the effect of different changes is different for different architectures:

From 0.5.0 to 0.6.1nd, the threaded code model changed from classical direct or indirect threaded code to primitive-centric direct threaded code. In addition, on IA32 the top-of-stack is no longer kept in a register (without explicit register allocation); registers are scarce on IA32, and without explicit register allocation GCC then spills the stack pointer to memory, causing a significant slowdown compared to not keeping the top-of-stack in a register.

On the IA32 CPUs, switching to primitive-centric direct-threaded code buys a speedup, because it eliminates the cache consistency problems these CPUs have with classical direct threaded code (where code fragments are close to data) [Ert02, Section 3], and which shows up in some of these benchmarks, especially cd16sim. Interestingly, the AMD64 versions of Gforth 0.5.0 outperform the IA32 versions on the same machine, even though the AMD64 versions have no architecture-specific

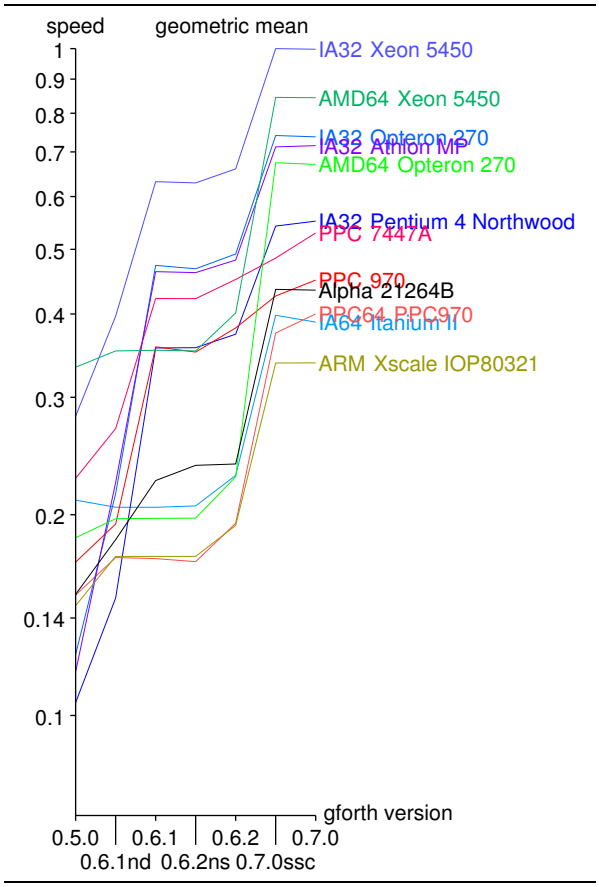


Figure 3: Performance per cycle, geometric mean of benchmarks (without brainless) of the best-compiled versions on all machines

tuning at all. Classical direct threading showed a benefit on the small benchmarks we usually use during development, but obviously these small benchmarks are not representative of large application benchmarks.

Most other machines also show an improvement from going to primitive-centric direct threaded code, because they usually used indirect threaded code in Gforth 0.5.0, and direct-threaded code is faster on most architectures.

From 0.6.1nd to 0.6.1: This enables dynamic superinstructions with replication on several architectures (Alpha, IA32, PPC), and gives large speedups on these machines. On architectures that we did not have available for testing when releasing 0.6.1 (AMD64, ARM, IA64, PPC64), this feature is not supported (it requires architecture-specific code for maintaining cache consistency) and therefore there is no change between 0.6.1nd and 0.6.1 on these architectures.

From 0.6.1 to 0.6.2ns There are no new performance features, so performance should be the same between these variants, and it generally is; we have no good explanation for the speedup on the Alpha 21264B machine.

From 0.6.2ns to 0.6.2 27 static superinstructions were enabled. They buy a small speedup even on systems where dynamic superinstructions work, because the native code for a static superinstruction is optimized compared to the equivalent dynamic superinstructions, which just consists of a concatenation of the code of its parts. Static superinstructions buy a larger speedup on systems where dynamic superinstructions are not supported, because there the static superinstructions also buy a part of the benefit that the dynamic superinstructions give otherwise: fewer NEXTs and better branch prediction. Looking at the individual benchmarks, static superinstructions help most of the benchmarks, but lexex is not affected.

From 0.6.2 to 0.7.0ssc there are a number of new performance features, with different effects on different architectures:

Several architectures (AMD64, ARM, IA64, PPC64) became available for testing, and now Gforth supports *dynamic superinstructions with replication* on them; note how AMD64 and PPC64 now catch up to the performance of IA32 and PPC on the same machines.

Automatic tuning: The build script automatically tests whether Gforth works when built with explicit register allocation and/or a C type for double-cell integers, and enables these features if they work (i.e. in the usual case). Explicit register allocation gives significant speedups on IA32 and AMD64.

Branch target alignment inserts padding in the native code such that the targets of branches are aligned to cache line boundaries. This provides a significant speedup on the Alpha; this feature is also implemented for IA32 and AMD64 (but with padding limited to 1 byte), but we have seen little effect there (we also tried more padding).

We also added *workarounds for GCC performance bugs*, resulting in more GCC versions having good performance. This does not show up much in these graphs, which show only the binary from the best-performing GCC, but it is responsible for much of the speedup on PPC: For Gforth 0.6.2, the best-performing GCC for PPC was 2.95, and it performs similarly for Gforth 0.7.0, but there gcc-4.3 performs a little better.

We have also implemented *faster mixed-precision division*, but we do not think that this shows up in these benchmarks.

From 0.7.0ssc to 0.7.0, multiple-state static stack caching is enabled. Unfortunately, on most architectures GCC cannot use more than one register for this purpose; so in addition to always keeping one stack item in a register, Gforth 0.7.0 can now also keep no stack item in a register, and switch between these two states to minimize the work needed. In theory this improves the performance for se-

quences like `! 5`, but as we can see, for most architectures (except PPC and PPC64) there is no speedup in application benchmarks.

On PPC and PPC64, GCC can use enough registers for keeping up to 8 stack items in registers, and up to 3 registers are useful [EG05], and that's what Gforth 0.7.0 uses on these architectures; static stack caching provides a speedup then. We suspect that there are also enough registers usable on IA64 and SPARC, but have not tested this.

3.3 Architectures and machines

We can also look at Fig. 3 to compare architectures and machines.

If you look for the best-performing system for running Gforth, the Xeon 5450 performs best per cycle among the machines we tested. In addition, it also has the highest clock rate, so it has the best absolute performance.

Another interesting question is whether to use 64-bit (AMD64, PPC64) or 32-bit (IA32, PPC) binaries of Gforth if you do not need 64-bit cells. In theory there is a speed advantage on AMD64 over IA32, because AMD64 has more registers available; unfortunately GCC makes no productive use of these registers when compiling Gforth; performance disadvantages of the 64-bit versions are the doubled memory requirement for all cells, including the threaded code, resulting in more cache misses; also, on the Xeon 5450 (and Core 2, but not on Opteron/Athlon 64), decoding is a little slower in 64-bit mode. On PPC64, there is no register advantage and no decoding slowdown.

Looking at the results, the 32-bit versions beat the 64-bit versions. There are some differences between the benchmarks here: cd16sim and fcp show the same performance in both architectures on the Opteron, but on Xeon the 32-bit architecture is a little faster (probably due to the decoding slowdown). For benchgc and lexex, the slowdown of the 64-bit version is significant (more than a factor of 1.2). This may be caused by the benchmarks doing something differently depending on cell size. E.g., for benchgc the cell size may change when and how often garbage collection is called. Or it could be a result of more cache misses.

For the PPC970, there is a slowdown in the 64-bit version even for cd16sim and fcp. One reason for that could be that we had fewer GCC versions available for PPC64 than for PPC; however, gcc-4.1 performed well for PPC and was available for PPC64, so we are not very confident that this explanation is correct. Unfortunately, we don't have any other explanation.

Another remarkable thing is how close the performance of the IA32 Opteron is to the IA32 Athlon MP; this confirms that the K8 (Opteron, Athlon 64) is really mostly a 64-bit variant of the K7 (Athlon MP, XP).

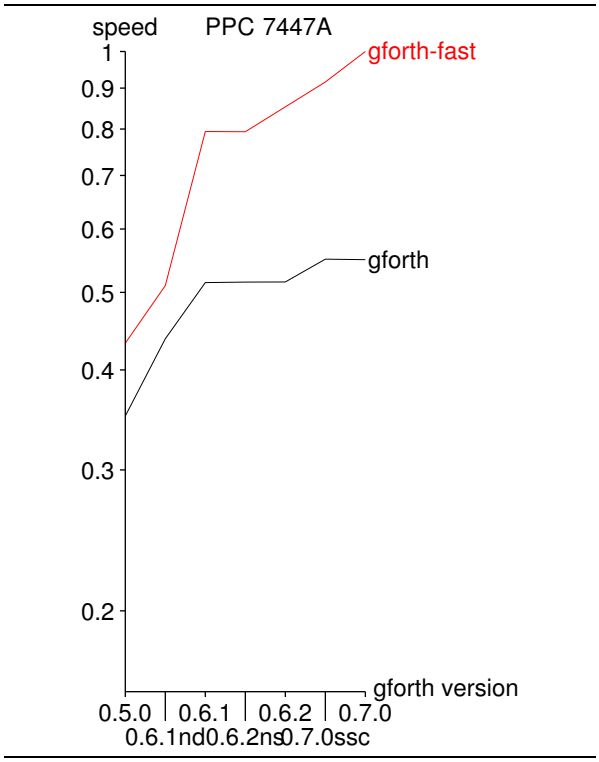


Figure 4: Benchmarking vs. debugging engine

Another interesting result is that all IA32 and AMD64 machines beat all the others in performance per cycle in Gforth 0.7.0; even the Pentium 4, which has a well-deserved reputation for raising the clock rate at the cost of lower performance per cycle beats all the other architectures.

This is probably due to the indirect branch predictors of these CPUs rather than the architecture itself; and these branch predictors benefit from dynamic superinstructions with replication. Even though dynamic superinstructions reduce the number of executed NEXTs (and thus the number of executed indirect branches) by a factor of more than 3, there are still a lot of indirect branches executed, and they cost a lot unless correctly predicted.

You can see this effect especially well by looking at the PPC7447A line and comparing it to the IA32 lines. In Gforth 0.5.0 and 0.6.1nd, it is the runner-up machine (after the Xeon) in performance-per-cycle, but with the enabling of dynamic superinstruction and replication, it is passed by the Opteron and Athlon MP, and the Pentium 4 also comes close. Finally, it is passed by the Pentium 4 with the enabling of explicit register allocation in Gforth 0.7.0 (PPC has enough registers that GCC performs good register allocation even without explicit register allocation).

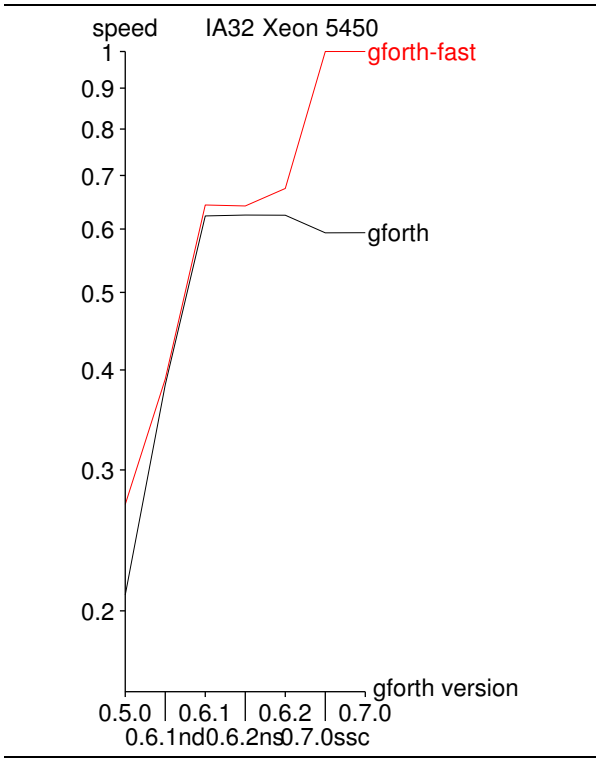


Figure 5: Benchmarking vs. debugging engine

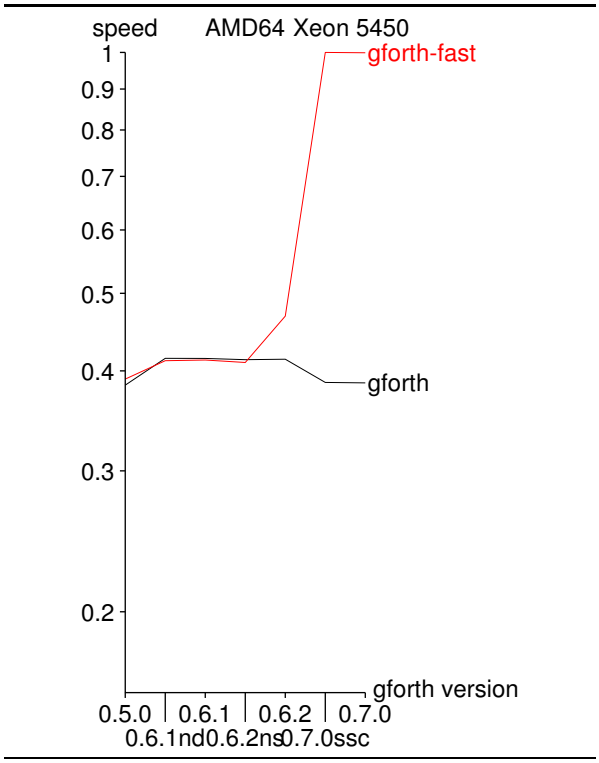


Figure 6: Benchmarking vs. debugging engine

3.4 Gforth-fast vs. gforth

Gforth comes with two engines: the debugging engine `gforth` and the benchmarking engine `gforth-fast`. The debugging engine performs some actions that cost performance, and it disables various performance features to allow better error reporting. How much does this cost, and has it changed over time, and why?

Figure 4 shows the graph for the PPC7447A. Already in Gforth 0.5.0, the debugging engine is slower, because it maintains a copy of the IP and RP virtual machine registers in memory (to allow better error reporting on invalid memory accesses etc.).

Both benefit to a similar amount from switching from indirect-threaded code to primitive-centric direct threaded code in 0.6.1nd; there is also a change in the way that IP is maintained that has no obviously visible effect on the PPC7447A, and is therefore explained later for a machine where the effect is visible.

Gforth-fast benefits a little more from dynamic superinstructions with replication in 0.6.1, probably because before it stalled longer waiting for the branches to resolve (whereas gforth was still busy maintaining IP and RP). There is no change in 0.6.2ns, as expected.

In 0.6.2, gforth-fast gains static superinstructions and a corresponding speedup, whereas the debugging engine does not enable static superinstructions in order to be able to report at which primitive an exception occured.

Both engines benefit from improvements in 0.7.0ssc (for this machine probably from GCC performance bug workarounds). On this machine gforth-fast profits from the more sophisticated stack caching in 0.7.0, whereas this stack caching is disabled in the debugging engine to support better reporting of stack underflows.

While the graphs for most other machines can be explained in a similar way, there are a few interesting deviations:

Figure 5 shows the graph of the IA32 Xeon. For gforth-0.5.0, IP is maintained in memory by using a global variable for it, which requires loading it at every access. Starting from gforth 0.6, IP is kept in a register, but is stored to memory on every instruction boundary. This eliminates the loads and also guarantees that the in-memory IP always points to a primitive. Apparently the stores alone are very cheap[1], resulting in performance for the debugging engine from 0.6.1nd to 0.6.2ns that is very close to the performance of gforth-fast. On other IA32 machines the performance of the debugging engine is actually slightly higher for these versions, but we

[1] Loads alone are also relatively cheap, but round trips through memory are usually expensive.

have no explanation for that.

Gforth 0.7.0 does not automatically tune the debugging engine to use explicit register allocation (to make building Gforth more robust and faster), so in the step from 0.6.2 to 0.7.0ssc we see the speedup from explicit register allocation in gforth-fast, but no speedup in gforth.

The slowdown for the debugging engine from 0.6.2 to 0.7.0ssc is due to workarounds for GCC performance bugs. These workarounds do have a cost; they pay for themselves on many compiler versions, but on the ones that don't need them they still cost.

Figure 6 shows the graph of the AMD64 Xeon. Unlike IA32, we have no classical direct threading with its cache consistency problems and also no spilling of SP, so the performance changes very little from 0.5.0 to 0.6.1nd. In addition, GCC manages to avoid loading IP from memory in 0.5.0 (resulting in code like for 0.6.1nd).

Dynamic superinstructions with replication are disabled in Gforth 0.6 on AMD64, so we see no speedup from that, and a flat line for the debugging engine until 0.6.2. In 0.7.0ssc one would expect dynamic superinstructions with replication to take effect, and they do for gforth-fast, but not for the debugging engine. The reason is that the debugging engine accesses a global variable (the saved IP) in every primitive, and on AMD64 global variables are referenced in a PC-relative way. This makes each primitive non-relocatable, effectively disabling dynamic superinstructions with replication for the debugging engine on AMD64.

3.5 GCC versions

All the graphs until now only showed the performance with the best-performing GCC version. Here we look at how well the different gforth-fast versions perform on different GCC versions on a few different architectures.

Figure 7 shows the graph for the PPC7447A. Gforth 0.5.0 and 0.6.1nd do not perform any optimizations that are broken by newer GCC versions, so their lines are relatively flat. Gforth 0.6.1–0.6.2 gain performance by using dynamic superinstructions with replication and work around GCC performance bugs up to gcc-3.3, but gcc-3.4 (released in 2004, i.e., after Gforth 0.6.2) and later introduced new performance bugs that disable dynamic superinstructions in these versions. Gforth-0.7.0 works around these performance bugs successfully, but in doing so apparently falls pray to a gcc-3.2 performance bug that disables dynamic superinstructions with replication. The GCC version that works best across all Gforth versions is gcc-2.95.

Figure 8 shows the graph for the IA32 Xeon. Again, gcc-2.95 shows the best performance across the board, and is the only compiler that builds

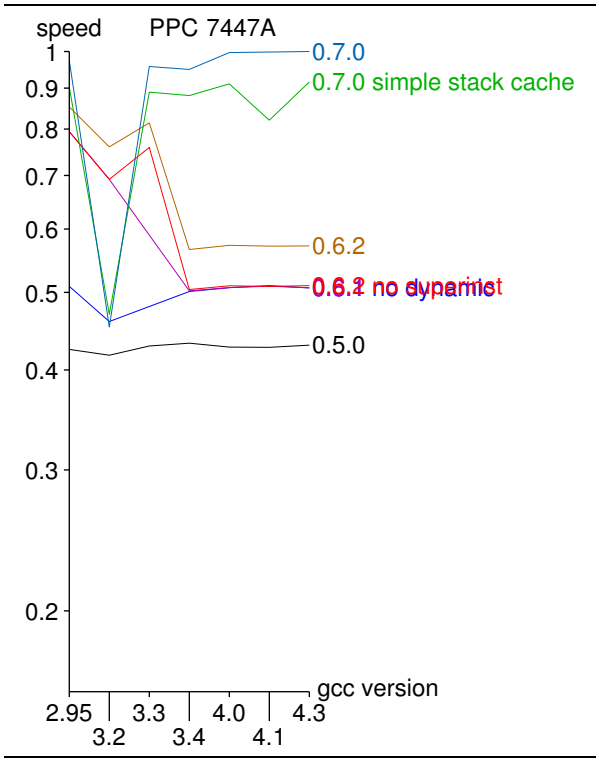


Figure 7: Gforth versions on different GCC versions

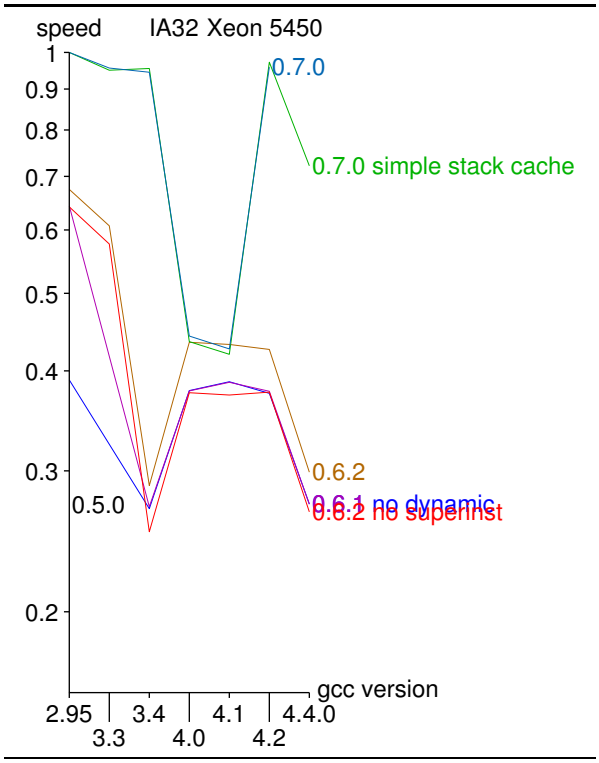


Figure 8: Gforth versions on different GCC versions

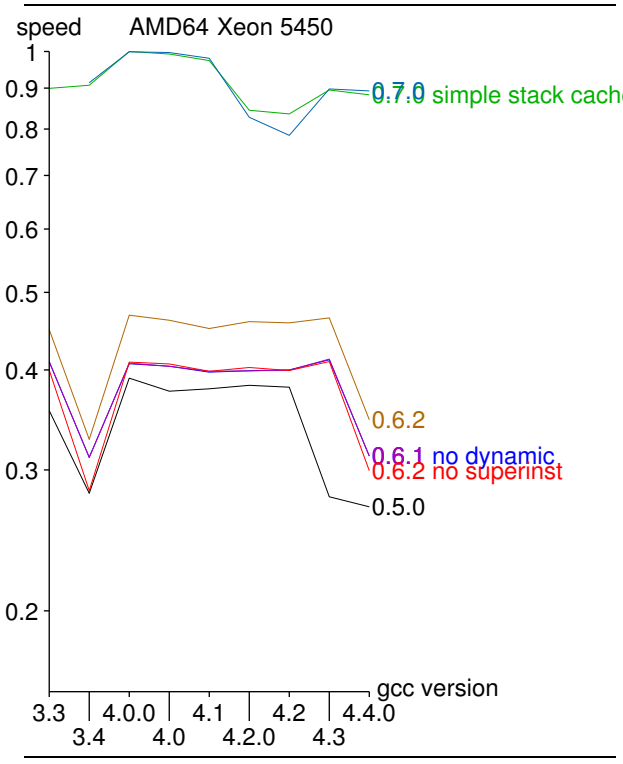


Figure 9: Gforth versions on different GCC versions

Gforth 0.5.0. Gcc-3.3 gratuitiously changed the code order, breaking gforth-0.6.1 as a result. We worked around this problem in 0.6.2.

The workarounds in Gforth-0.6.2 for GCC performance bugs work up to gcc-3.3 and then fail. Gcc-3.4 is particularly bad in sharing one indirect branch for all the NEXTs, completely disabling the branch predictor of the CPU (GCC PR15242); that bug also causes the slowdown of 0.6.1nd on gcc-3.4. Gcc-4.0–4.2 fixed this bug, restoring at least a part of the performance, but the PR15242 problem is back in gcc-4.4.0, giving us bad performance again.

Gforth 0.7.0 successfully works around the performance bugs having to do with code ordering and indirect branches in GCC $\leq$ 4.3, but gcc-4.0 and 4.1 spill important virtual machine registers, hurting performance. In addition to resurrecting PR15242, gcc-4.4.0 (released after Gforth-0.7.0) features a new (or worsening) performance bug that makes NEXT longer and slower, resulting in the slowdown shown in the graph. This performance bug uncovered a bug in the implementation of static stack caching in Gforth 0.7.0 (and that bug is responsible for there being no result for 0.7.0 with static stack caching and gcc-4.4.0).

Figure 9 shows the graph for the AMD64 Xeon. Unfortunately, gcc-2.95 is not available for AMD64. Gforth $\leq$ 0.6.2 does not use dynamic superinstructions with replication on AMD64 anyway, so the lines for these Gforth versions run mostly in parallel, reflecting the presence of PR15242 in gcc-3.4 and 4.4.0, and their absence in the other version, with one exception: gcc-4.3 exhibits the PR15242 problem for gforth-0.5.0, but not for gforth-0.6.x.

Gforth-0.7.0 successfully works around the GCC bugs that disable dynamic superinstructions with replication. The cause for the performance variations between the gcc-4.x versions seems to be a performance bug that makes NEXT longer (and slower) in varying amounts between these versions.

4 Future work

This work uncovered some performance issues (in particular the unnecessarily long NEXT) that we plan to work around.

In addition, there are some performance ideas that we plan implement, in particular inlining [GE04].

Finally, this performance evaluation should be enhanced by comparing Gforth with other Forth systems. One challenge here is finding a large enough set of application benchmarks that run on all Forth systems.

5 Related work

Instead of working around GCC bugs as we do, one could also fix GCC. Prokopski and Verbrugge [PV08] propose a good method for letting GCC preserve the order of basic blocks and similar assumptions that are helpful for implementing code-copying optimizations like dynamic superinstructions. They don't just disable or restrict optimizations; they record the basic block order at the start and then restore it at the end (if possible), or report an error (if not).

6 Conclusion

The performance of default-compiled Gforth has improved a lot between Gforth 0.5.0 (2000) and 0.7.0 (2008), typically by a factor of 3.

The most significant factor for that performance improvement is the introduction of dynamic superinstructions with replication. While that was relatively easy to implement as a prototype, making it work on a wide range of architectures and GCC versions is a larger effort: First, it requires a small amount of architecture-specific code; more significantly, new GCC versions often break this feature, requiring programming workarounds for these performance bugs. So while this feature was introduced in Gforth 0.6.x, in many practical cases (e.g., various Debian packages) it was disabled in these versions. Gforth 0.7.0 includes a lot of work to make this feature more widely available.

There are also many other performance features, but they often only have a small effect (e.g., static superinstructions) or only on one or a few architectures (e.g., automatic tuning to enable explicit register allocation, which helps a lot on IA32). The combined effect of all these optimizations is quite significant, though.

Another interesting result is that Gforth has proven to be very portable, with even the very old Gforth 0.5.0 running on architectures and being compiled with compilers that did not exist when it was released.

References

[EG03a] M. Anton Ertl and David Gregg. Implementation issues for superinstructions in Gforth. In *EuroForth 2003 Conference Proceedings*, 2003.

[EG03b] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN '03 Conference on Programming Language Design and Implementation*, 2003.

[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.

[EG05] M. Anton Ertl and David Gregg. Stack caching in Forth. In *21st EuroForth Conference*, pages 6–15, 2005.

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

[FW86] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, March 1986.

[GE04] David Gregg and M. Anton Ertl. Inlining in Gforth: Early experiences. In *EuroForth 2004 Conference Proceedings*, pages 33–40, 2004.

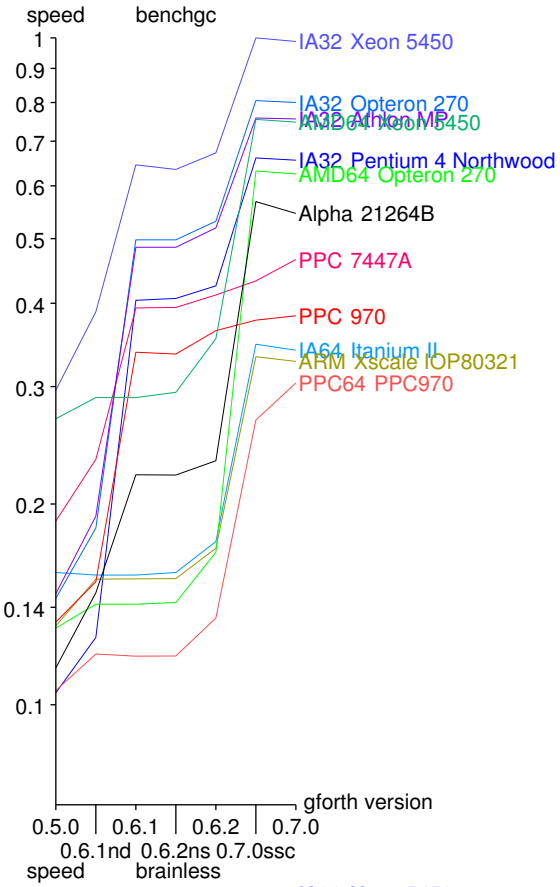
[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

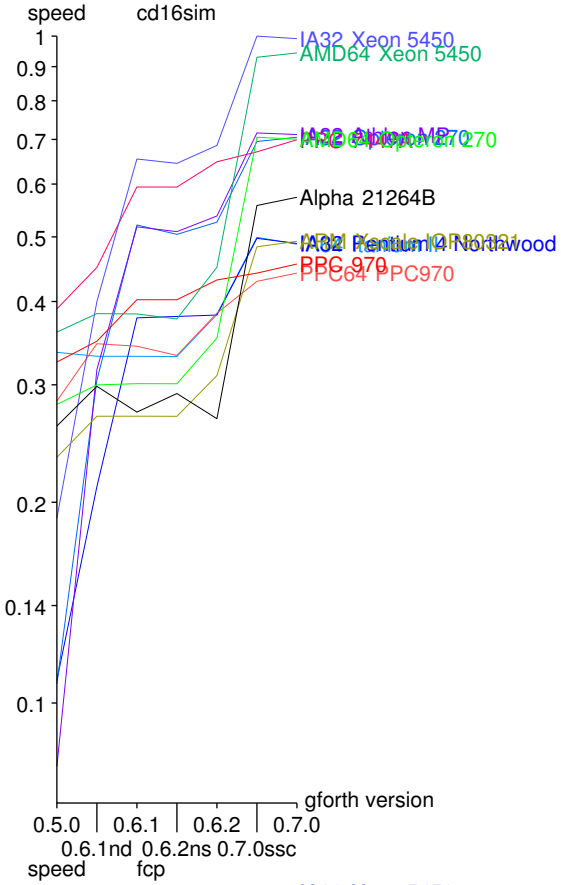
[PV08] Gregory B. Prokopski and Clark Verbrugge. Compiler-guaranteed safety in code-copying virtual machines. In *Compiler Construction (CC'08)*, pages 163–177. Springer LNCS 4959, 2008.

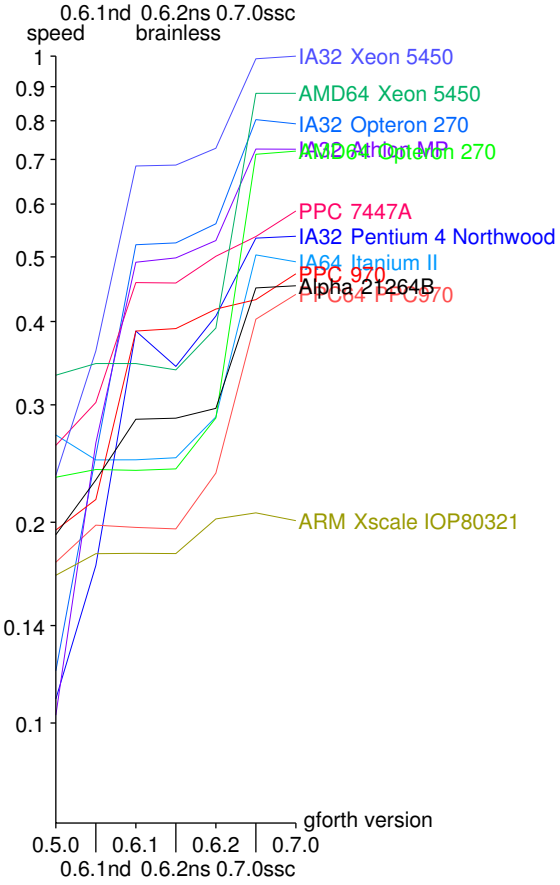
[RS96] Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Faculty of Information Technology, Helsinki University of Technology, May 1996.

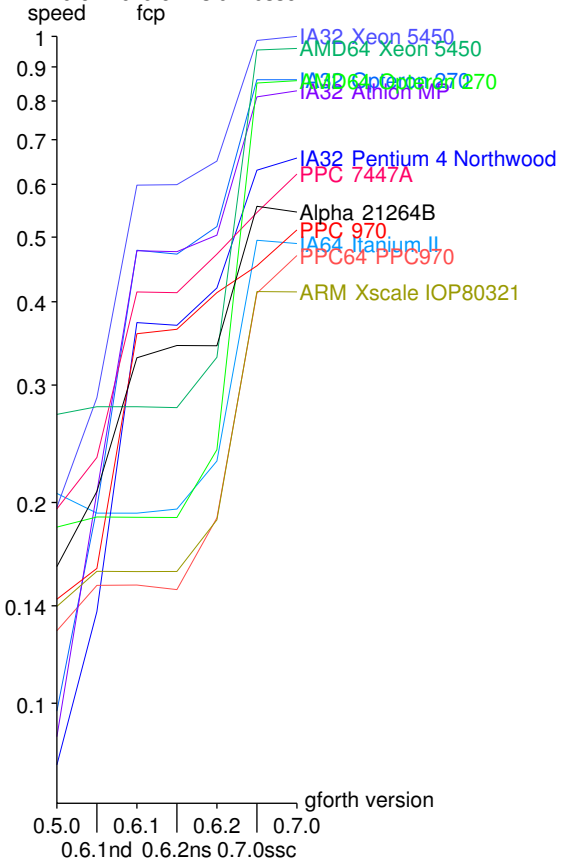
A Extra data

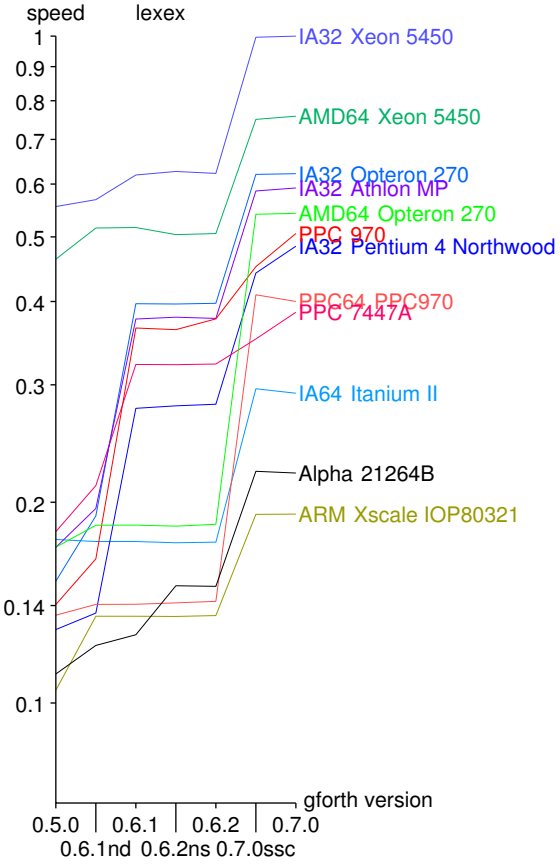
A.1 Individual benchmarks

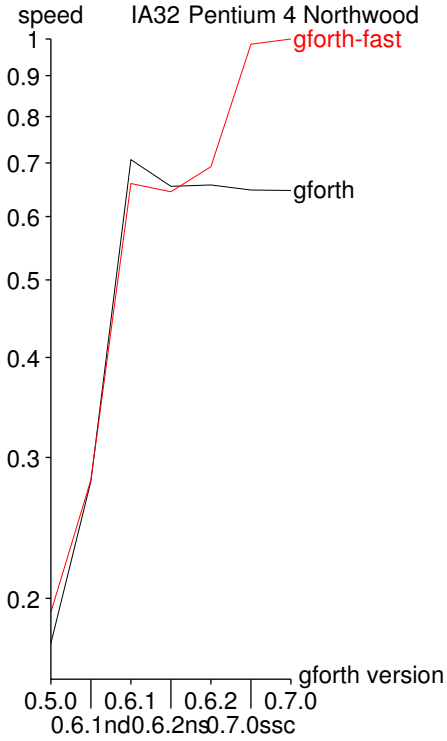




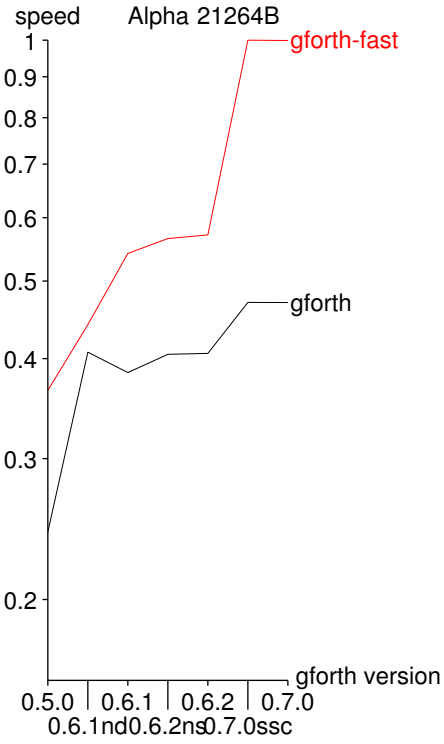


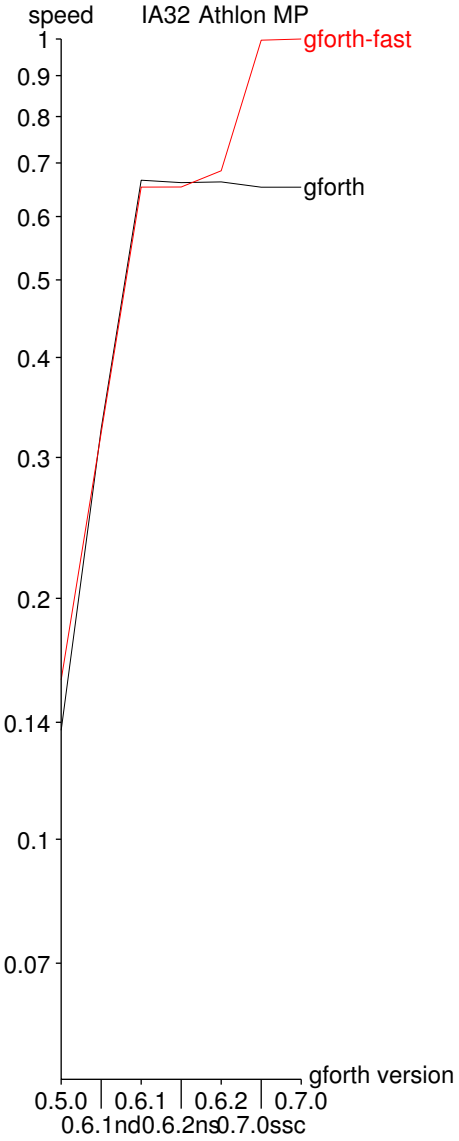


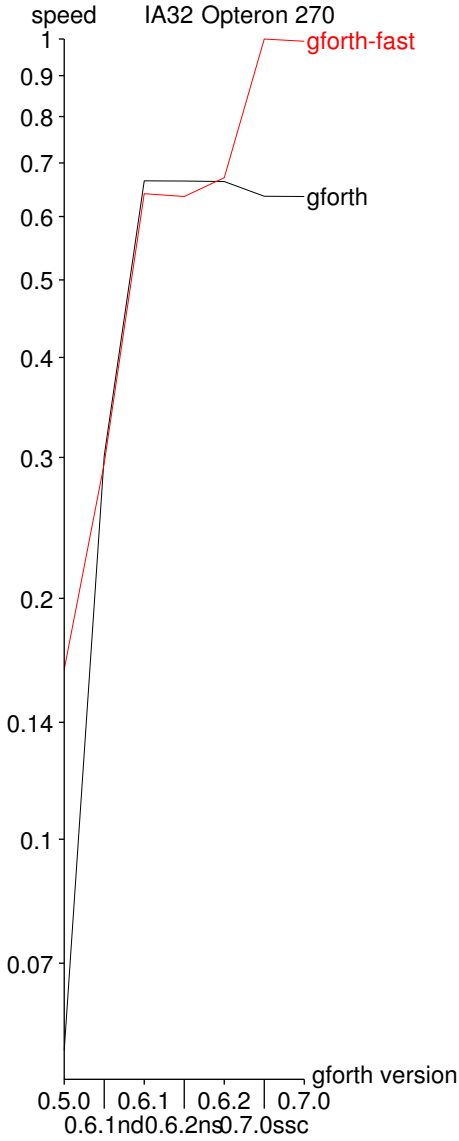


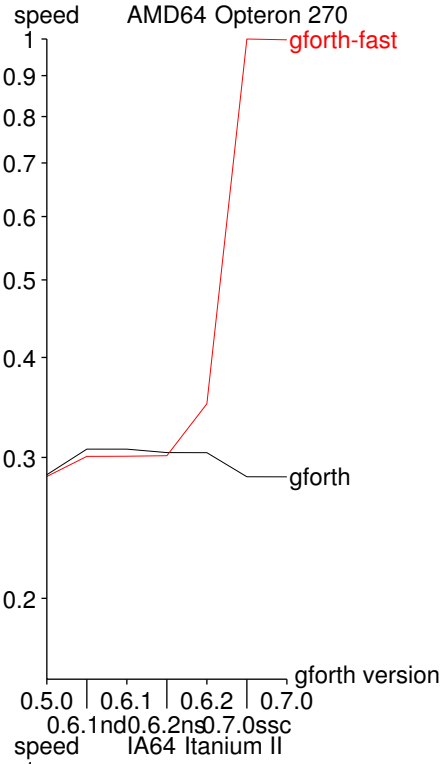


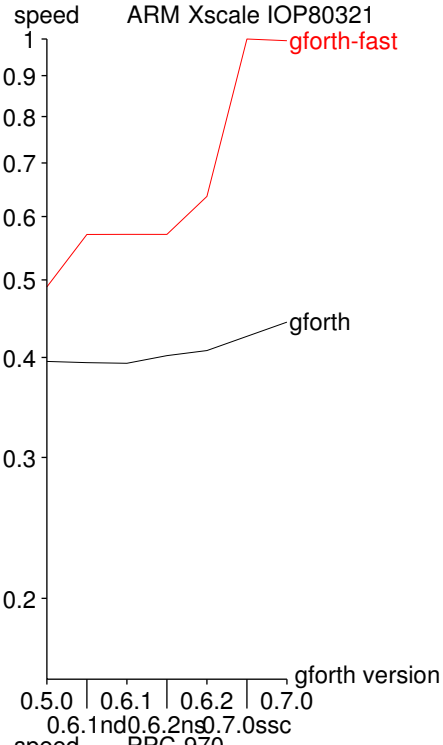
A.2 Debugging vs. benchmarking engine

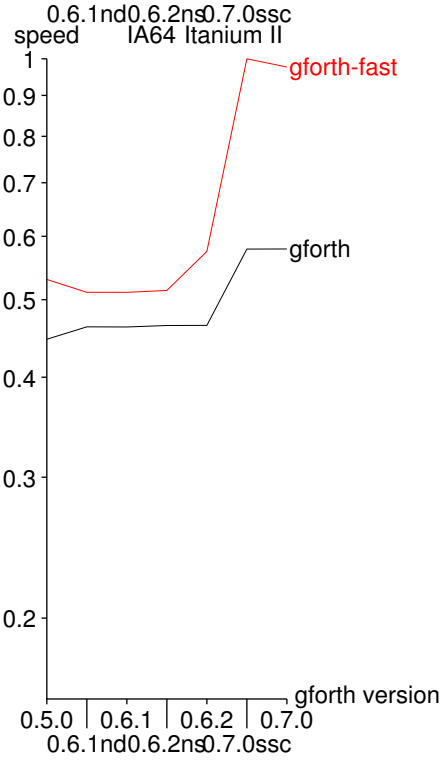


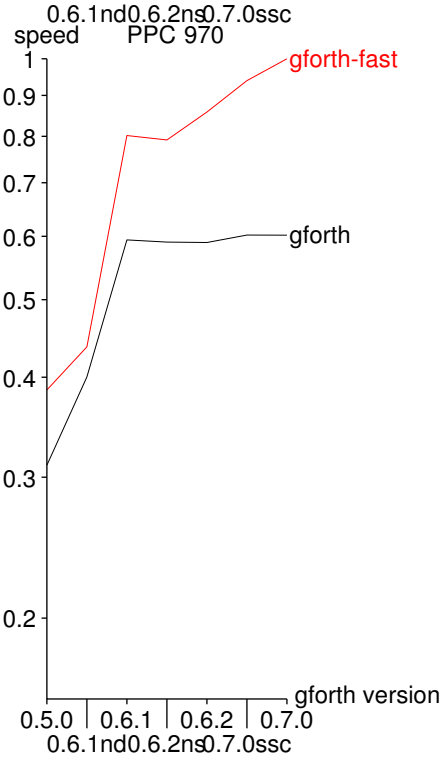

speed
IA32 Athlon MP
gforth-fast
gforth
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.14
0.1
0.07
gforth version
0.5.0
0.6.1nd
0.6.1
0.6.2ns
0.6.2
0.7.0ssc
0.7.0

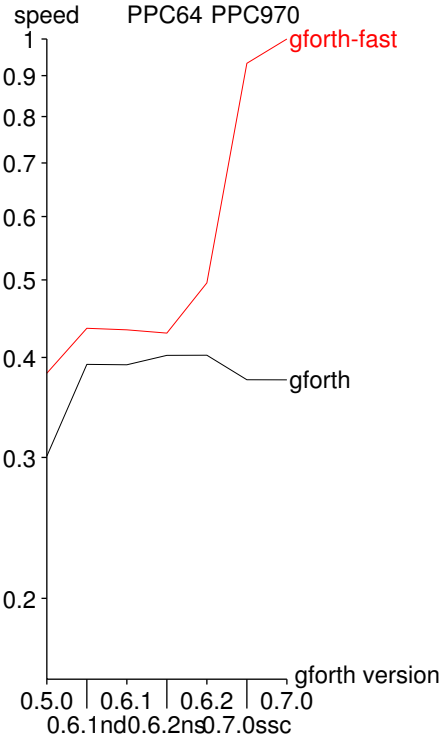

speed
IA32 Opteron 270
gforth-fast
gforth
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.14
0.1
0.07
gforth version
0.5.0
0.6.1nd
0.6.1
0.6.2ns
0.6.2
0.7.0ssc
0.7.0

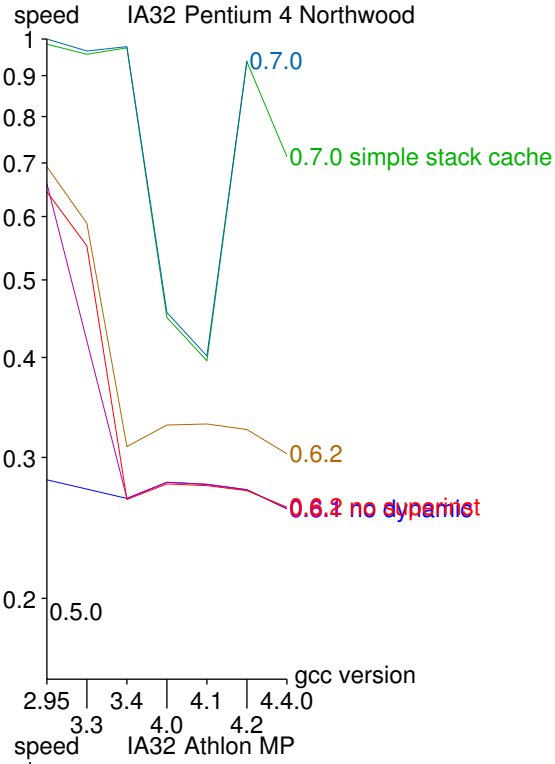

speed
AMD64 Opteron 270
gforth-fast
gforth
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
gforth version
0.5.0
0.6.1
0.6.2
0.7.0
0.6.1nd
0.6.2ns
0.7.0ssc

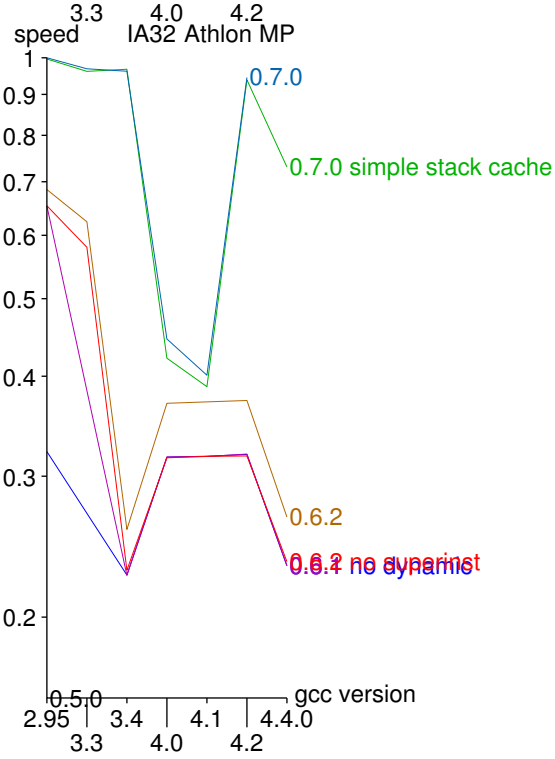

speed
ARM Xscale IOP80321
gforth-fast
gforth
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
gforth version
0.5.0
0.6.1
0.6.2
0.7.0
0.6.1nd
0.6.2ns
0.7.0ssc


speed
IA64 Itanium II
gforth-fast
gforth
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
gforth version
0.5.0
0.6.1
0.6.2
0.7.0
0.6.1nd
0.6.2ns
0.7.0ssc

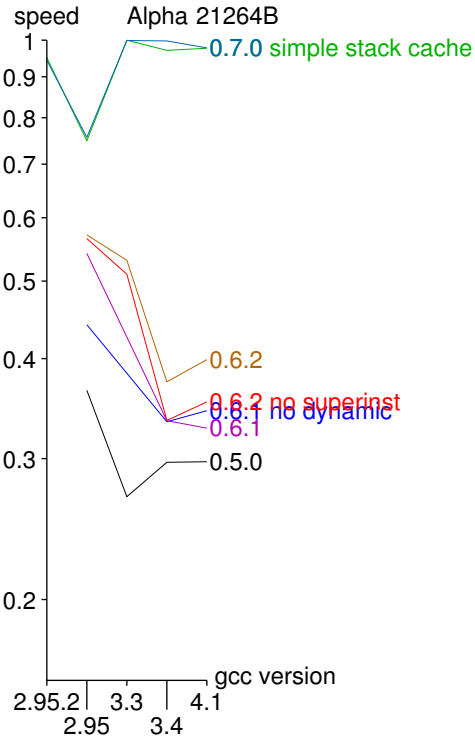

speed
PPC 970
gforth-fast
gforth
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
gforth version
0.5.0
0.6.1
0.6.2
0.7.0
0.6.1nd
0.6.2ns
0.7.0ssc

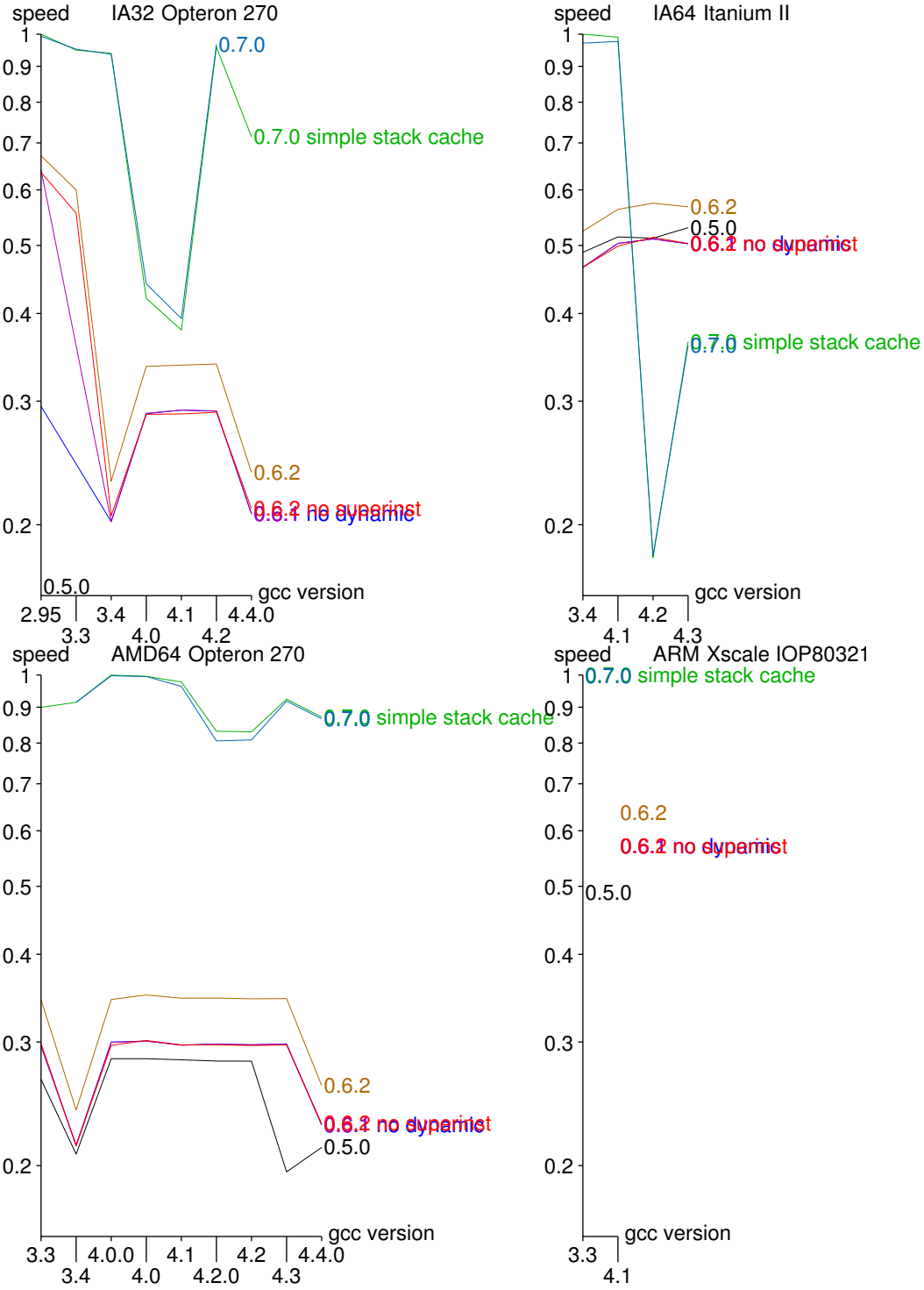


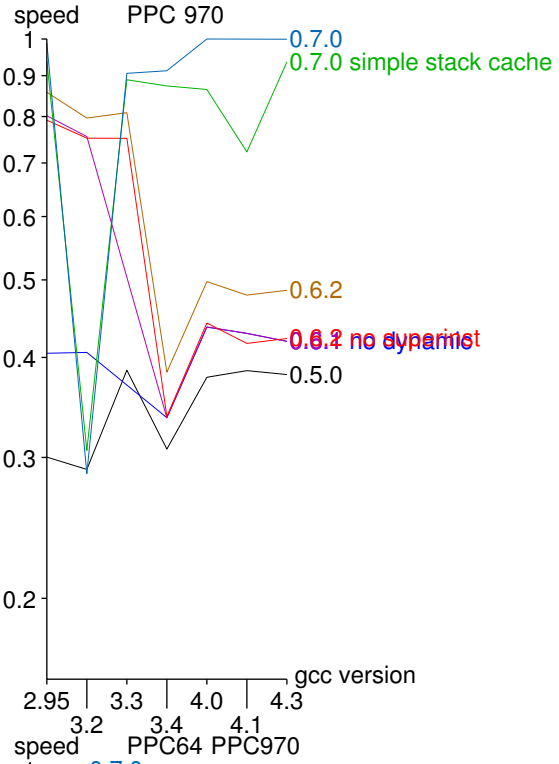


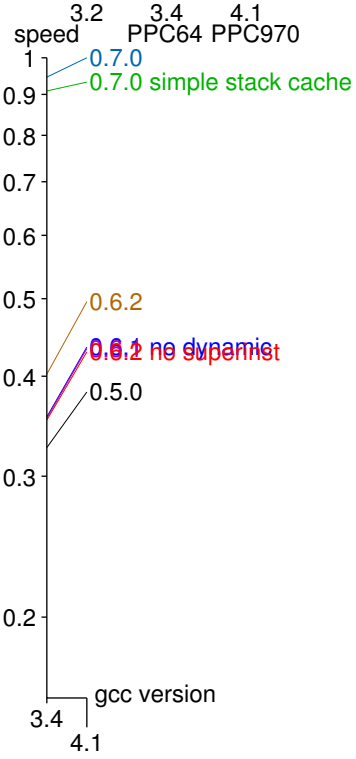


A.3 GCC versions




speed
IA32 Opteron 270
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.7.0
0.7.0 simple stack cache
0.6.2
0.6.2 no dynamic
0.6.1 no dynamic
0.5.0
gcc version
2.95
3.3
3.4
4.0
4.1
4.2
4.4.0
speed
IA64 Itanium II
0.6.2
0.5.0
0.6.2 no dynamic
0.7.0 simple stack cache
gcc version
3.4
4.1
4.2
4.3
speed
AMD64 Opteron 270
0.7.0 simple stack cache
0.6.2
0.6.2 no dynamic
0.5.0
gcc version
3.3
3.4
4.0.0
4.0
4.1
4.2.0
4.2
4.3
4.4.0
speed
ARM Xscale IOP80321
0.7.0 simple stack cache
0.6.2
0.6.2 no dynamic
0.5.0
gcc version
3.3
4.1





A.4 Forth systems