ABI-CODE: Increasing the portability of assembly language words

M. Anton Ertl*
TU Wien

David Kühling

Abstract

Code words are not portable between Forth systems, even on the same architecture; worse, in the case of Gforth, they are not even portable between different engines nor between different installations. We propose a new mechanism for interfacing to assembly code: abi-code words are written to comply with the calling conventions (ABI) of the target platform, which does not change between Forth systems. In the trade-off between performance and portability, abi-code provides a new option between code words and colon definitions. Compared to code words, the abi-code mechanism incurs an overhead of 16 instructions on AMD64. Compared to colon definitions, we achieved a speedup by a factor of 1.27 on an application by rewriting one short colon definition as an abi-code word.

1 Introduction

Code words are not portable between Forth systems, even between Forth systems running on the same architecture[1]. The main reason for that is that there are no standard registers for the stack pointers.

For Gforth[2], the situation is even worse: Because it uses GCC to build its inner interpreter, and GCC decides the register allocation on its own, code words are not even portable between Gforth installations[3] and engines (in particular, not between gforth and gforth-fast).

In this paper, we describe the new abi-code facility of Gforth that allows writing code in assembly language that is portable between different Gforth installations and engines. If other Forth systems implement this facility, too, it could enable porting assembly language code to other Forth systems running on the same platform.

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

[1]We will use this notion of portability between Forth systems running on the same architecture in the rest of this paper.

[2]Gforth is a fast and portable Forth implementation. It achieves portability by for creating the machine code of the primitives with a C compiler.

[3]In particular, Gforth 0.6.2 compiled with one version of GCC is not necessarily compatible with the same Gforth version compiled with another version of GCC; also, Gforth 0.6.2 configured with explicit register allocation (--enable-force-reg) is not necessarily compatible with the same Gforth version configured without this option. These problems should be less frequent in Gforth 0.7.0, because there explicit register allocation is tried by default.

2 Basic Idea

2.1 abi-code

Abi-code words are called according to the calling convention of the platform, passing and returning the stack pointers through parameters. The calling convention is usually described in the application binary interface (ABI) documentation of the platform, leading to the name abi-code.

The data-stack pointer is passed as first parameter, and is returned as result. An address to a memory cell containing the FP-stack pointer is passed as second parameter, and the FP-stack pointer is returned by storing the changed value in this memory cell; if the FP stack is not accessed, the second parameter can be ignored. In C terms, an abi-code word has the following prototype:

```
Cell *word(Cell *sp, Float **fp_pointer)
```

The stack layout and the sizes of the stack items are also relevant: In Gforth both data and FP stack grow towards lower addresses, and the sizes of the items on the stack are the same as in memory (i.e., 1 cells and 1 floats).

Here is an example of using abi-code on Linux-AMD64[4]:

```
abi-code my+  ( n1 n2 -- n3 )
\ SP passed in rdi, returned in rax
lea rax,[rdi+8] \ new sp in result reg
mov rdx,[rdi]   \ get old tos
add [rax],rdx   \ add to new tos
ret             \ return from my+
end-code
```

To make our examples easier to read, we present them in Intel syntax (destination first) rather than the syntax of the Gforth assembler. You can find a version of this example in Gforth syntax (so you

[4]Unfortunately, Windows uses a different convention on AMD64

can try it out) in the Gforth manual (development version[5]).

Most calling conventions pass the parameters in registers, but IA-32 calling conventions usually pass them on the architectural stack, and therefore require slightly more overhead:

```
abi-code my+
mov eax,4[esp] \ sp in result reg
mov ecx,[eax]  \ tos
add eax,#4     \ update sp (pop)
add [eax],ecx  \ sec = sec+tos
ret            \ return from my+
end-code
```

And here is an example of an FP `abi-code` word on Linux-AMD64:

```
abi-code my-f+
mov  rdx,[rsi]       \ load fp
fld  qword ptr[rdx] \ r2
add  rdx, 8         \ update fp
fadd qword ptr[rdx] \ r1+r2
fstp qword ptr[rdx] \ store r
mov  [rsi],rdx      \ store new fp
mov  rax,rdi        \ sp in result reg
ret                 \ return from my-f+
end-code
```

Here we have some extra overhead, because the FP stack pointer `fp` is passed in and out in a memory location; also, here we do not need to update the data stack pointer `sp`, so moving `sp` to the result register requires a separate instruction.

Unlike ordinary `code` words, `abi-code` words need a special routine to invoke them out of a threaded-code inner interpreter. The definition of `abi-code` in `gforth-fast` on Linux-AMD64 is equivalent to the following:

```
: abi-code ( "name" -- )
  create also assembler
;code ( ... -- ... )
\ doabicode routine
mov   [r15],r14    \ 1)store TOS to memory
mov   rdi,r15      \ 2)sp to 1st arg reg
movsd [r12],xmm8   \ 1)store FP TOS
lea   rax,[r9+10H] \ 3)body of name
mov [rsp+6b0H],r12 \ 2)store fp in memory
lea rsi,[rsp+6b0H] \ 2)2nd arg: fp address
call  rax          \ 4)call name's body
mov rdx,[rsp+6b0H] \ 2)load fp
mov   r14,[rax]    \ 1)load TOS
mov   r15,rax      \ 2)copy sp to sp reg
movsd xmm8,[rdx]   \ 1)load FP TOS
mov   r12,rdx      \ 2)copy fp to fp reg
NEXT               \ 5)threaded dispatch
end-code
```

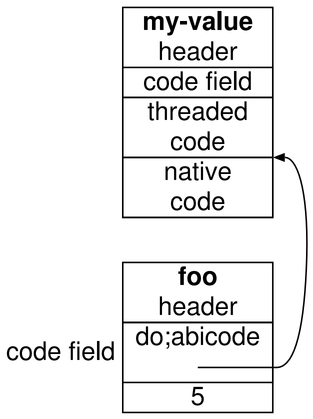

Figure 1: Code field layout for `;abi-code`-defined words

The `doabicode` routine consists of the following components (the numbers in the comments above refer to the component numbers):

1. Saving the tops of the stacks to memory to comply with the memory-stack convention of `abi-code` words; and loading the tops of the stacks back into registers afterwards. This is needed because `gforth-fast` keeps the tops of both the data stack and the FP stack in registers (r14 and xmm8).
2. The FP stack pointer is stored in memory, and argument registers are set up. And after the call the new stack pointers are moved to the registers of sp (r15) and fp (r12).
3. The address of the called routine is computed (it starts at the body address, which is computed by adding 2 cells (10H) to the CFA in r9.
4. The actual call.
5. Invoke the next primitive (NEXT).

Gforth also contains a primitive `abi-call` that is used for invoking `abi-code` words (primitive-centric code [Ert02][6]); it has the same components except that the address of the called routine is found as immediate argument of the primitive, not through the CFA.

This overhead will probably be a little lower in native-code compilers, but most of the components will be there too.

[5]`http://www.complang.tuwien.ac.at/forth/gforth/cvs-public/`

[6]In primitive-centric threaded code every non-primitive (colon definition, constant, etc., and `abi-code` words) is compiled to a primitive followed by an immediate argument.

2.2 ;abi-code

;abi-code is to ;code what abi-code is to code. The routine after ;abi-code is passed a third parameter: the body address of the word for which the routine provides the behaviour. In C terms, the routine has the following prototype:

```
Cell *word(Cell *sp, Float **fp_pointer,
           char *body)
```

Here is an example of using ;abi-code on Linux-AMD64:

```
: my-value ( w "name" -- )
  create ,
;abi-code ( -- w )
  \ sp in rdi, address of fp in rsi
  \ body address in rdx, temp reg: rcx
  lea rax,[rdi-0x8] \ new sp in return reg
  mov rcx,[rdx]     \ load value from body
  mov [rax],rcx     \ store value on stack
  ret
end-code

5 my-value foo
```

Unlike for ;code routines, we cannot use the start address of this routine as code address in the CFA of an indirect-threaded Forth. Instead, we have to use a solution like the one we use for does>-defined words: The code address points to a routine do;abicode that calls ;abi-code routines. It finds the address of the routine to call in the cell right after the CFA (see Fig. 1). Gforth has two cells for the xt field, the second being used for does>-defined and ;abi-code-defined words. The do;abicode routine and the ;abi-code-exec primitive contains components similar to the doabicode routine shown above.

3 Discussion

3.1 Compared to code

The main disadvantage of abi-code words compared to code words is that they have additional calling overhead. We will look at the performance difference resulting from this overhead in Section 4.

The advantage of the abi-code approach is that it provides a simple and stable interface.

For programmers the advantage of that stability is that their assembly language words are portable between Gforth engines (gforth, gforth-fast, gforth-itc), and portable across installations (in particular, indepedent of the GCC version used). If abi-code was implemented by other Forth systems, abi-code words could be written to be portable across systems.

For the system implementor, the advantage of the stable interface is that the system is not tied to using the same register assignments internally forever. It can change, e.g., the number of stack items in registers, or move the data stack pointer to a different register if that results in faster code on a new processor.

The simplicity helps programmers by not having to learn and remember top-of-stack registers that the system may use internally; and it helps the system implementor by not having to teach that to programmers. Indeed, Mitch Bradley once commented that he went back from keeping the TOS in a register to keeping all stack items in memory in order to provide a simpler interface to the users. Abi-code provides the same benefit at a lower cost: we pay the additional overhead only when executing an abi-code word, not in the whole system.

3.2 Why use the ABI?

Why did we choose to use the ABI? Couldn't the same benefits not be achieved with another approach?

The major reason we chose to use the ABI is that it is easy to get GCC to generate an ABI call. There are some other benefits, however:

- We can use this facility to call functions written in non-assembly languages that conform to the ABI; these functions would have to be written to access the stacks, though. Indeed, we will probably modify the implementation of the libcc C interface [Ert07] to use this mechanism rather than the less refined calling mechanism it uses now.
- For each platform, the ABI is already given, so the system implementor does not need to decide what the interface should be. As long as there is only one system involved, this is not a particular advantage, but as soon as several systems implement a common interface, they would have to standardize on a common interface for each platform they support, and as anybody witnessing a standards process knows, that tends to be rather time-consuming. And that's the best case; instead, each vendor might go their own way on a new platform, so the advantage of a common interface would disappear.

There is also a disadvantage to using the ABI: ABIs often require passing the stack pointers in ways that are suboptimal. E.g., in our AMD64 example, the data stack pointer is passed in in a different register than it is passed out, and on IA-32 it is even passed in through memory; and the limitations of common ABIs have led us to pass the

FP stack pointer through memory in any case (see Section 3.4).

3.3 Alternatives

An alternative would be to have conventional `code` words, but supply macros that switch from the system's register-and-stack setup to a fixed register setup and back, just like the `NEXT` macro invokes the next word, whether the system is direct-threaded, indirect-threaded or subroutine-threaded.

In terms of overhead for a given interface this approach would save relatively little compared to an `abi-code`-like implementation: Only the computation of the call target, the call itself and the return would be eliminated.

One reason why we did not choose this approach is that we have no good way to get the register allocation for Gforth's engines out of GCC and into these macros; in contrast, with `abi-code` GCC does the setup of the call and the restoration for us.

Similarly, we could use `code` words, but abstract away from the concrete register allocation of a Forth system and the concrete implementration of the stack by providing macros for accessing the stacks and/or for the logical registers (e.g., stack pointers, temporary registers); if several systems implement the same macros, code may be portable between them even if their register allocation differs.

A problem with this approach is that some systems keep the top-of-stack in a register and others in memory. On most architectures you cannot use a memory access wherever you can use a register, so it would be tricky to set up the macros such that they can be implemented on all systems. Moreover, we again cannot use this approach for Gforth, because we have no automatic way to get the actual register allocation out of GCC and into these macros. And, to achieve true portability, Forth vendors would have to agree on the macros (and these may be architecture-specific, e.g., how many temporary registers are usable by `code` words), whereas someone else has already standardized the ABI.

Another approach that might be implementable in Gforth would be to do the setup and call in C code with `asm()` statements. This would allow us to use an arbitrary interface, not limited by the ABI.

A problem of this approach is that there is no guarantee that GCC plays along; it could run into a situation where it cannot allocate registers and would then fail to build Gforth. Or it could produce an abysmal register allocation that slows down Gforth significantly (that actually happens often enough without us playing such games).

The benefit of this approach over `abi-code` words does not appear to be big enough to merit the effort of implementing it.

3.4 What parameters and how to pass them

Gforth has four stacks visible to the engine: data, return, FP, and locals stack. Moreover, there is the instruction pointer (IP). Which of these pointers should be passed to the called word?

We decided to pass only the data and FP stack pointers (sp and fp), because these are the stacks normally used for dealing with general-purpose and floating-point data. The other stacks and IP are typically used for implementing Forth-system internal stuff like control flow or locals. Most users of `abi-code` will probably not want to implement such words; passing and returning them would increase the cost of executing every abi-code word, so we decided not to pass them.

How do we pass these two stack pointers and how do we return them? At first we passed sp and fp as parameters, and returned them in a struct, leading to the following prototype:

```
struct ac_ret {Cell *sp; Float *fp;};
struct ac_ret word(Cell *sp, Float *fp);
```

However, when we looked at the generated code, we found that this is implemented inefficiently on most platforms: The calling convention on most platforms returns a struct by storing it to memory before returning and loading it from memory in the caller (*pcc calling convention*). So, with two stack pointers in the struct this costs two stores and two loads. And, what's more, the programmer would have to write these stores and have to deal with the target address for this struct.

There is at least one platform (Linux-AMD64), where the standard calling convention passes small structs in registers, and on such platforms this could be the most efficient way of passing the stack pointers, but unfortunately GCC generates inefficient code (redundant stores) even on that platform.

So overall, while this could be an efficient method, in practice it isn't. And on most platforms it is cumbersome to use.

Therefore, we decided to switch to the currently-used way to pass the stack pointers:

```
Cell *word(Cell *sp, Float **fp_pointer);
```

The downside here is that fp is passed and returned in a cumbersome way; but at worst this leads to as many loads and stores as returning a struct on platforms with the pcc calling convention, and in most cases (no FP stack access, or unchanged FP stack depth) it will have fewer loads and/or stores. This approach is also easier to learn and to use for programmers, especially for words that don't access the FP stack.

We also considered several other approaches, but did not implement them:

- Put both pointers in memory and pass pointers to them; as a variation, put them in a structure in memory and pass one pointer to that. The main advantage would be that both stack pointers would be passed in the same way, leading to a cleaner interface. The disadvantage is that this approach is less efficient and requires more loads (and usually stores) than our chosen approach.

- Another, mostly orthogonal option would be to have different words for different usages: E.g., we could have `abi-code-sp`, where only sp is passed and returned (avoiding the overhead of storing fp to memory and loading it back); and maybe `abi-code-fp`, where only fp is passed and returned (in the same way that sp is passed now). This would increase the efficiency, but it would also increase the complexity, implementation and documentation effort of the interface, so we decided not to take this approach for now.

- A reviewer suggested passing the word's arguments and returning the result directly as defined in the calling convention, rather than through the Forth stack. The called routine could then also be called from C in the familiar way without having to set up a memory area for the stack and passing a pointer to that. This would require generating a wrapper that automatically translates between the Forth arrangement and the calling convention. We have done such a thing for calling C functions in the libcc C interface [Ert07], and this approach could also be used here.

  But we don't think that this would be very useful, for the following reasons: 1) In a C interface we usually want to call pre-existing C routines, whereas here the typical usage will be to write new assembly code for this specific problem, so the exact kind of parameter passing convention does not make a big difference. 2) Most calling conventions provide no good way to pass back several results. 3) The wrapper would probably incur extra overhead; e.g., transferring the parameters from the Forth stack to the C stack on IA-32, where it is just as hard to access. 4) One could not implement words such as `roll` with such a mechanism.

For `;abi-code`, we have to pass the body address of the child word in addition to sp and fp. We just pass it as extra parameter.

3.5 Other Forth systems

`Abi-code` solves a problem of Gforth. Would there be a benefit to implementing `abi-code` in other Forth systems? Yes:

- Code using `abi-code` would be portable between Forth systems (on the same platform), unlike code using `code`. This could also be achieved by agreeing on a standard interface to `code` words for each platform, but reaching such an agreement can be a long and arduous process. For the ABI somebody else went through that process, so if we use that, we save ourselves that effort. As for the disadvantages, using the ABI leads to more overhead when executing `abi-code` words. What's worse, on some architectures there are different ABIs for different operating systems, so `abi-code` words do not necessarily port to other operating systems, even on the same architecture and Forth system, unlike `code` words on most systems.

- The infrastructure used for implementing `abi-code` can also be used for calling functions in other languages that use the ABI. However, most systems already have a more convenient C interface that does not require the called function to access Forth stacks. Still, given that these Forth systems implement the ABI for the C interface, it should be easy to implement `abi-code` on them.

There are additional requirements to make code portable across Forth systems: The stacks have to grow in the same direction in the systems (in Gforth all stacks grow downwards).[7] And the stack items have to have the same size and format; that's not a problem for cells, but different Forth systems use different FP formats/sizes on IA-32 (64-bit vs. 80-bit floats).

4 Performance

4.1 Benchmarks

The benchmarks are written for Gforth. We measure both `gforth-fast --no-dynamic` (direct threaded code) as well as the default `gforth-fast` (with various optimizations). Other systems use different implementation techniques, with different effects on performance, so take these results with a grain of salt.

We compare different ways to implement `1+`. This word is so short that its cost is relatively minor compared to the overheads of the various implementation techniques, so the overheads should dominate.

[7] We could get around that requirement by having macros for accessing the stack at a certain depth.

Also, we can implement `1+` both as simple words, or through defining words. We complement this micro-benchmark with a result from an application (Section 4.4).

We compare four different ways of defining simple words:

primitive Primitives come with Gforth, and Gforth knows quite a bit about them, in particular, how to use them in dynamic superinstructions [RS96, PR98, EG03a]. And Gforth can also perform other optimizations on them [Ert02, EG04, EG05]. These optimizations do not include combining a sequence of `1+ ... 1+` into `n +`, however.

code-def A `code` definition. Gforth knows very little about such words, so these are executed as direct-threaded code.

abi-code-def `Abi-code` definitions are usually executed through a primitive `abi-call`; all Gforth optimizations can be applied to this primitive, but the called routine is executed as-is.

colon-def A simple colon definition. Gforth does not perform inlining (yet). Colon definitions are invoked through the primitive `call`. But `gforth-fast` optimizes the body of the colon definition with a static superinstruction [Ert02] for the sequence `lit +`.

We also compare the corresponding four ways of defining `1+` through defining words:

field-def Using the built-in field definition word `+field`; children of this word are compiled to a primitive `lit+`, which has all the usual optimizations applied. This primitive uses a literal constant in the threaded code, so it has a little more overhead than the primitive `1+`.

;code-def To maintain the primitive-centric code [Ert02] in Gforth, the child of such a word cannot be compiled directly to threaded code like code-def. Therefore Gforth uses a primitive `lit-execute` to invoke it, adding some overhead; in particular, there is an additional indirect branch (from the primitive to the code after `;code`). Moreover, the other indirect branch will always be mispredicted in some of our benchmark setups: those where we use dynamic superinstructions and run on CPUs with BTBs.

;abi-code-def Children of a `;abi-code` word are executed through a primitive `;abi-code-exec` similar to `abi-call`.

```
' 1+ alias primitive
\ add    rbx,0x8 \ increment IP
\ add    r14,0x1 \ increment TOS (gcc way)
\ next primitive or NEXT

code code-def
 add rbx,0x8     \ increment IP
 inc r14         \ increment TOS
 jmp [rbx-0x8] \ NEXT
end-code

abi-code abi-code-def
\ ABI: SP passed in rdi, returned in rax
 mov rax,rdi        \ sp into return reg
 inc QWORD PTR[rdi] \ increment TOS
 ret
end-code

: colon-def 1 + ;

\ indirect definitions through defining a
\ defining word

1 0 +field field-def drop
\ add r14,[r9+0x10] \ >body @ +
\ add rbx,0x8       \ increment IP
\ NEXT

: my-field1 ( n -- )
    create ,
;code ( n1 -- n2 )
 \ sp=r15, tos=r14, ip=rbx, cfa=r9
 add rbx,0x8       \ increment IP
 add r14,[r9+0x10] \ >body @ +
 jmp [rbx-0x8]     \ NEXT
end-code
1 my-field1 ;code-def

: my-field2 ( n -- )
    create ,
;abi-code ( n1 -- n2 )
 \ sp in rdi, returned in rax,
 \ addr of fp in rsi, body address in rdx
 mov rcx,[rdx] \ fetch increment from body
 mov rax,rdi   \ sp into return reg
 add [rdi],rcx \ add increment to TOS
 ret
end-code
1 my-field2 ;abi-code-def

: my-field3 ( n -- )
    create ,
  does> ( n1 -- n2 )
    @ + ;
1 my-field3 does>-def
```

Figure 2: Benchmark definitions (Intel syntax for assembly)

does>-def Children of `does>` words are compiled to be invoked using the primitive `does-exec` (which is similar to a sequence of `lit` and `call`).

Figure 2 shows the definitions of these words for the Linux-AMD64 platform.

The benchmark consists of a loop that contains a sequence of these implementations of `1+`. We measure a loop with a sequence of 23 `1+`s, and subtract the time for a loop with 3 `1+`s. This gives the time for executing 20 `1+`s without the loop overhead or startup effects. Note that these micro-benchmarks are unrealistic in their branching behaviour and therefore give unrealistic branch prediction results.

4.2 Machines

The performance of these benchmarks is influenced strongly by how well indirect branches are predicted and by the cost of mispredictions when they happen. Therefore we measure the performance on two different CPUs:

Athlon 64 X2 4400+ This processor has a branch target buffer (BTB), which predicts (to the first order) that each indirect branch jumps where it jumped to the last time it was performed. The misprediction penalty is around 12 cycles.

Core 2 Duo E8400 This processor has a history-based indirect-branch predictor that is usually more accurate than a branch target buffer. The misprediction penalty is around 12 cycles.

All of these CPUs implement a return stack, so the returns at the end of `abi-code` words are predicted correctly.

We also vary the options used with the `gforth-fast` engine:

no-dynamic This is direct-threaded code.

default All optimizations are on. In particular, dynamic superinstructions benefits everything except `code-def` and `;code-def`; static superinstructions benefit `colon-def`; and static stack caching benefits `abi-code-def` and `;abi-code-def`.

4.3 Results

Figure 3 shows the results for direct-threaded code, and Fig. 4 shows the results for optimized code. For both machines, we show instructions, some data about branches and branch mispredictions, and cycles. The metric we actually care about on a particular platform is the cycles, but the other metrics are also interesting, because they help explain the cycle counts that we see, and can help understand what performance to expect on other machines or for other benchmark settings.

For cycles and instructions, the count per executed word (implementation of `1+`) is shown; branches and branch mispredictions are scaled up by a factor of 10, for two reasons: to make their size better visible; and to reflect the approximate cost of branch mispredictions in cycles.

Cycles and Instructions

The instruction counts are the same between the machines, because the same binaries are executed on both machines.

For threaded code (Fig. 3), we see that the primitive has a similar cyle and instruction counts as `code-def`; actually, the instruction count and cycle count is slightly better for the hand-written code word compared to the gcc-generated primitive.

Executing the `abi-code` word is more expensive by 13 instructions and 8–9 cycles; for the optimized code, the difference is 11 instructions and 4–7 cycles. This means that one will still use `code` words where their portability disadvantage is acceptable and the number of dynamically executed instructions in the word is relatively small on average (several dozen instructions or less).

The colon definition performs a similar number of instructions (and cycles) as the `abi-code` word for this micro-benchmark, but that's because `1+` is such a tiny word. For words with more functionality, a colon definition in a threaded-code Forth will require more instructions (and cycles) compared to a primitive or code word by a factor of 5-10 in many cases, whereas the `abi-code` word will only have the same 11–13 instructions of overhead as for this benchmark, not an overhead proportional to the functionality.

The instruction counts for `field-def`, `;code-def`, `;abi-code-def`, and `does>-def` are slightly higher than for the corresponding simple words (they fetch the increment from memory), but are otherwise similar to their corresponding simple words.

Branches and Mispredictions

Branch mispredictions have a strong influence on the cycle count, and the mispredictions in these micro-benchmarks are not representative of typical applications, so we have measured the number of branches and branch mispredictions, and present the results here.

For the branches, the Core 2 can count indirect branches (and their mispredictions), whereas the Athlon 64 can count taken branches (and their mispredictions). For these benchmarks, both

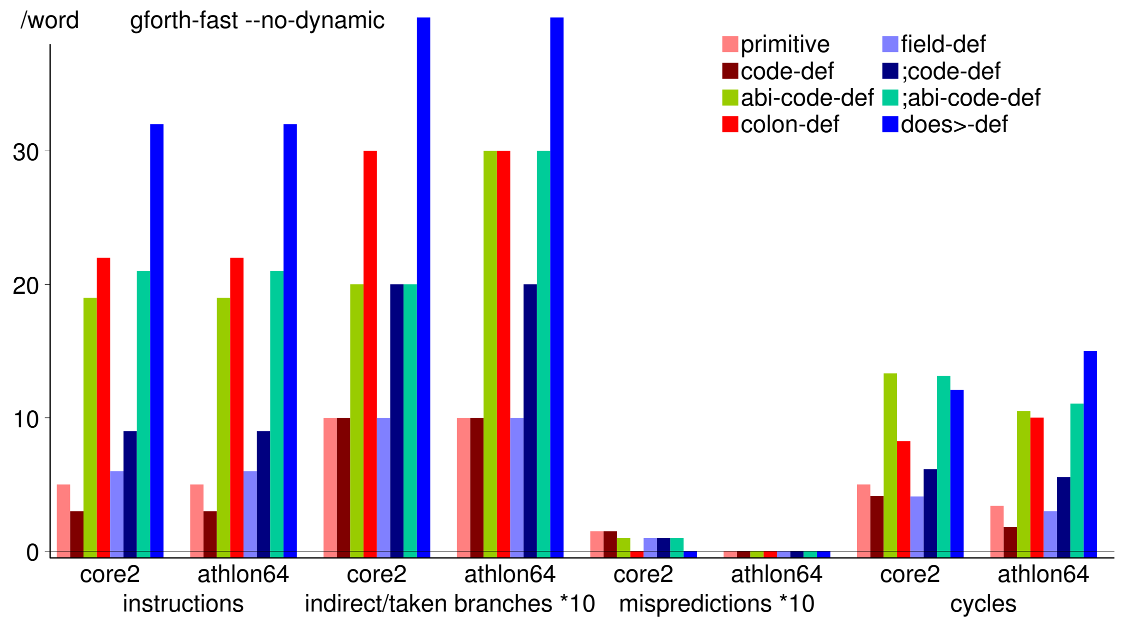

Figure 3: Performance results for `gforth-fast --no-dynamic` (direct threaded code)

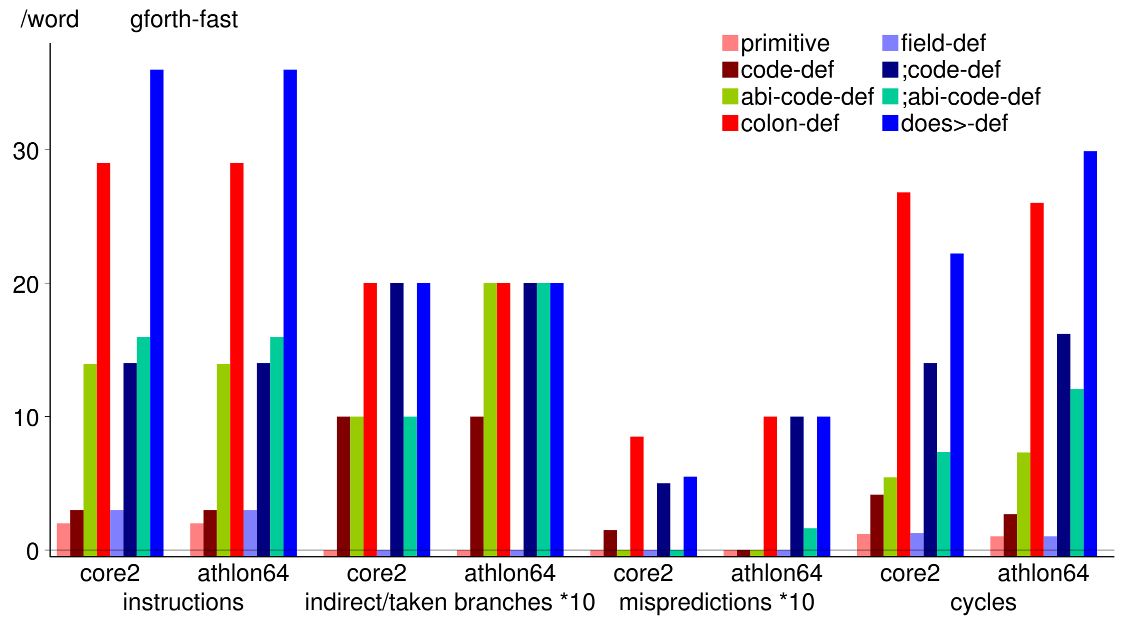

Figure 4: Performance results for `gforth-fast` default (with optimizations)

measurements result in the same branch counts in most cases, except for the `abi-code-def` and `;abi-code-def` cases: There the `ret` from the called word is counted as taken branch by the Athlon 64, but not as indirect branch by the Core 2. These returns are always predicted correctly by the return stack on both CPUs, so this difference does not affect the misprediction counts. The mispredictions differ between the CPUs, because they use different branch predictors.

Does the number of mispredictions differ systematically between `code` words and `abi-code` words? One difference is that `code` words cannot use the dynamic superinstruction optimization used for Gforth primitives, typically leading to more mispredictions than for primitives (but only partially in our micro-benchmark). For `abi-code` words, dynamic superinstructions can be applied to the `abi-call` primitive; and the indirect call inside this primitive will be well predictable using BTBs and more sophisticated predictors, because each instance of `abi-call` will always call the same code.

Even for this micro-benchmark the Core 2 behaves mostly as expected (for the optimizing

Gforth, see Fig. 4): The branch prediction accuracy is worse for `code-def` than for `abi-code-def`, resulting in a similar cycle count for both of these words.

For threaded code the situation is different: There primitives, `code`, and `abi-code` words all have to perform an indirect branch at the end of the word, and that branch will often ($\approx 50\%$ with a BTB) be mispredicted in real applications. Moreover, because there is only one replica of `abi-call` in a threaded-code system, the indirect call inside `abi-call` will also often be mispredicted if different `abi-code` words are used in the inner loop.

You may notice that the branch prediction accuracy on the Athlon 64 is better on this microbenchmark for threaded code than for the optimized version. That is an artifact of this microbenchmark; real-world code behaves differently [EG03b, EG03a].

4.4 Application performance

In a Mandelbrot set calculation program we replaced a short colon definition (7 words, straight-line code), with an `abi-code` word containing 11 MIPS instructions[8]. This resulted in a speedup by a factor of 1.27 on a 336MHz Ingenic XBurst Jz4720 running `gforth-fast --dynamic`. However, as with `code` words, this approach is only cost-effective if a significant part of the run-time is spent in one or a few words.

5 Related work

The classical Forth way to define words in assembly language is `code`...`end-code`. It has the disadvantage of being system-specific, or worse, in the case of Gforth, installation-specific.

Modern Forth systems also provide a C interface. The main use of this interface is to call libraries that have been developed independently, but it can also be used to call C functions written specifically for a Forth application; and it can be used to call such functions written in assembly language. However, these functions usually have to be compiled or assembled separately before loading the Forth system[9], in contrast to defining words in assembly language at the appropriate places in a Forth source file with `abi-code` and `code`.

New Micros' Max-Forth for the 68hc11 has a word called `code-sub` where the definitions have to end with an `rts` (return from subroutine) rather than a `jmp next` [Dum]. This avoids the need to hard-code the address of `next` and therefore increases the portability of hand-assembled machine code (there was not enough space for a Forth assembler). The implementation uses a run-time routine like Gforth does, but which is less elaborate than `doabicode` (no adjustment to an ABI necessary).

Looking beyond Forth, the Java Native Interface (JNI) [Lia99] shares a number of similarities with `abi-code`. It allows Java to call functions through an interface based on the calling conventions (ABI) combined with additional conventions. The called functions are portable across Java VM implementations, and even across platforms, if written in a portable language like C. There are also differences: JNI functions are compiled separately, and they are usually not written in assembly language.

6 Conclusion

`Abi-code` allows programmers to write assembly language words that work across Gforth engines and versions. If other Forth systems implement `abi-code`, too, they work even across Forth systems.

These words use the standard calling convention (ABI) of the platform, so they are easy to implement in Forth systems that are implemented with the help of a C compiler (like Gforth).

The price we pay for these advantages is an overhead of 11–13 instructions on AMD64 (4–9 cycles on current implementations) when invoking `abi-code` words. However, compared to colon definitions `abi-code` words can provide quite a bit of speedup (a factor of 1.27 by replacing one colon definition in one example application), at the cost of being architecture-specific. So `abi-code` provides a new option between colon definitions and `code` words in the tradeoff between performance and portability.

Acknowledgments

We thank the anonymous reviewers for their comments and suggestions, which helped improve the paper.

References

[Dum] Randy M. Dumse. *User Manual Max-FORTH*. New Micros.

[EG03a] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, 2003.

[8] `http://mosquito.dyndns.tv/freesvn/trunk/nanonote/forth/mandelbr.fs`

[9] Exception: Bernd Paysan used Gforth's libcc interface to generate the C code from Forth code upon loading, and that C code is compiled and linked right away.

[EG03b] M. Anton Ertl and David Gregg. The structure and performance of *Efficient* interpreters. *The Journal of Instruction-Level Parallelism*, 5, November 2003. http://www.jilp.org/vol5/.

[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 M. Anton Ertl, editor, *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.

[Ert07] M. Anton Ertl. Gforth's libcc C function call interface. In M. Anton Ertl, editor, *23rd EuroForth Conference*, pages 7–11, 2007.

[Lia99] Sheng Liang. *Java Native Interface: Programmer's Guide and Specification.* Addison-Wesley, 1999.

[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.

[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.