From `exit` to `set-does>` A Story of Gforth Re-Implementation

M. Anton Ertl[*]
TU Wien

Bernd Paysan

Abstract

We changed `exit` from an immediate to a non-immediate word; this requires changes in the deallocation of locals, which leads to changes in the implementation of colon definitions, and to generalizing `does>` into `set-does>` which allows the defined word to call arbitrary execution tokens. The new implementation of locals cleanup can usually be optimized to similar performance as the old implementation. The new implementation of `does>` has similar performance similar to the old implementation, while using `set-does>` results in speedups in certain cases.

1 Introduction

Over the years there were several complaints about not being able to tick `exit` in Gforth. In July 2015 we decided to do something about this. In combination with other innovations, this led to a number of further changes in the implementation, and eventually to a generalization of `does>`.

The story of these changes and the other implementation issues they touch on should be interesting and instructive for readers interested in Forth implementation techniques, and is told in Section 2. These changes were not performed for performance reasons, but performance should not suffer from them. In Section 3 we evaluate the performance impact with microbenchmarks. Section 4 discusses an implementation caveat for locals cleanup on native-code compilers.

2 The Story

While Forth-94 and Forth-2012 systems are allowed to implement `exit` as an immediate compile-only word, we have received a number of complaints about Gforth implementing `exit` this way, so we decided to change the implementation of `exit` into a non-immediate word in July 2015.

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

2.1 Locals cleanup

Now we had implemented `exit` as immediate compile-only word for a good reason: When `exit`ing a definition with locals, we need to remove the locals before exiting. In the following contrived example:

```
: foo { a } exit ;
```

the original immediate `exit` compiles `lp+ ;s`, where `lp+` increments the locals-stack pointer lp to remove `a` from the locals stack and `;s` returns to the caller of `foo`.

Our new, non-immediate `exit` is just an alias for `;s`, so we have to clean up the locals in some other way. We took the established approach of pushing additional data and an additional return address on the return stack. In our case the additional data is the depth of the locals stack at the start of the colon definition, and the return address points to a code fragment equivalent to

```
r> lp! ;s
```

except that we have a single primitive `lp-trampoline` that does what this sequence would do; the (sub-optimal) AMD64 code[1] for this primitive is:

```
mov  %rp,%rax
mov  0x8(%rp),%ip
lea  0x10(%rp),%rp
mov  (%rax),%lp
add  $0x8,%ip
mov  -0x8(%ip),%rdx
mov  %rdx,%rax
jmpq *%rax
```

So, an `exit` inside a colon definition with locals jumps to this code fragment, sets lp to its old value, and finally returns to the calling definition (see Fig. 1).

This solution for the clean-up problem poses the problem of where these additional return-stack

[1]Register names for virtual machine registers are replaced, as follows: `ip=rbx, rp=r13, lp=rbp, sp=r15, tos=r14, cfa=rcx`.

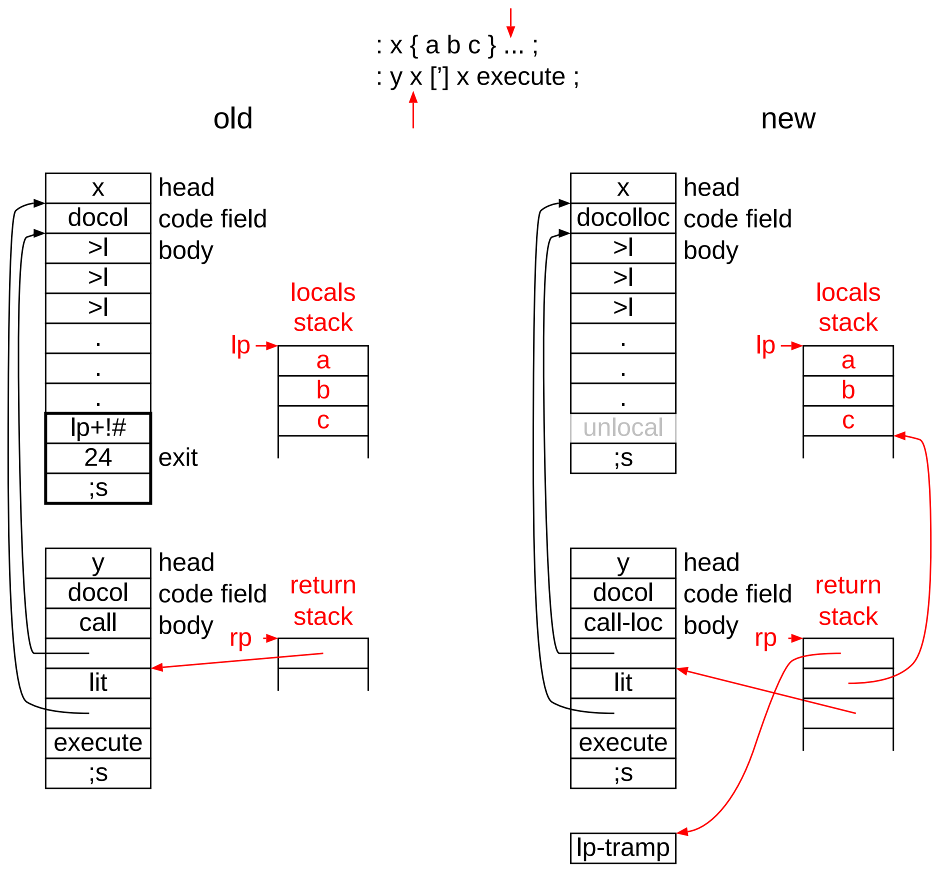

Figure 1: Old and new implementation of cleaning up locals; the state of the return and locals stack corresponds to execution being at the red arrows in the code.

items are pushed. A classical solution would be to do it at the first definition of locals. However, in Gforth, locals can be first defined inside control structures, e.g.:

```
: foo ?do { a } i loop ;
```

Either we push the return-stack items before the control structure, or we have to pop them off the return stack at the end of the loop[2].

We decided to push them before the control structure, on entering the colon definition, by changing the code field to point to a new routine `docolloc` instead of the ordinary `docol` routine. `Docolloc` peforms all the work that `docol` does, but in addition pushes the current value of lp and the address of the code fragment pointing to `lp-trampoline` on the return stack. Here you see both routines for the AMD64:

```
docol                     docolloc
mov   %rp,%rax            mov   %rp,%rax
mov   %ip,%rdx            mov   %ip,%rdx
lea   -0x8(%rp),%rp       lea   -0x18(%rp),%rp
mov   %rdx,-0x8(%rax)     mov   %rdx,-0x8(%rax)
                          lea   0x80(%rsp),%rdx
lea   0x18(%cfa),%ip      lea   0x18(%cfa),%ip
                          mov   %lp,-0x10(%rax)
                          mov   %rdx,-0x18(%rax)
mov   -0x8(%ip),%rdx      mov   -0x8(%ip),%rdx
mov   %rdx,%rax           mov   %rdx,%rax
jmpq  *%rax               jmpq  *%rax
```

Gforth uses primitive-centric threaded code [Ert02], so the routines `docol` and `docolloc` are executed only when the word is `execute`d or called through a `defer`red word. When calling the word directly from a colon definition (about 99% of the calls), Gforth uses the primitives `call` and `call-loc` that take (the body address of) the called definition from the next cell in the threaded-code:

[2] In general, whenever the locals stack becomes empty.

```
call                   call-loc
mov  %ip,%rdx          mov  %ip,%rdx
mov  (%ip),%ip         mov  (%ip),%ip
mov  %rp,%rax          mov  %rp,%rax
add  $0x8,%rdx         add  $0x8,%rdx
lea  -0x8(%rp),%rp     lea  -0x18(%rp),%rp
                       mov  %lp,-0x10(%rax)
mov  %rdx,-0x8(%rax)   mov  %rdx,-0x8(%rax)
                       lea  0x80(%rsp),%rdx
add  $0x8,%ip          add  $0x8,%ip
                       mov  %rdx,-0x18(%rax)
mov  -0x8(%ip),%rdx    mov  -0x8(%ip),%rdx
mov  %rdx,%rax         mov  %rdx,%rax
jmpq *%rax             jmpq *%rax
```

Gforth has an intelligent `compile,` that produces the appropriate primitive for the word, and it generates `call` for `docol` words, and `call-loc` for `docolloc` words, plus (in the next cell) the body address of the colon definition.

Some Forth programmers like to use code like `r> drop exit` to return to the next-but-one surrounding definition instead of the next one. If the next one uses locals, the programmer has to force a cleanup, and we provide the word `unlocal` to achieve this. So if the calling word uses locals, the sequence above has to be modified to `r> drop unlocal exit`. `Unlocal` just removes the additional return stack data and removes the locals from the locals stack:

```
unlocal              unlocal-;s
mov  %rp,%rax        mov  0x10(%rp),%ip
lea  0x10(%rp),%rp   mov  0x8(%rp),%lp
add  $0x8,%ip        add  $0x18,%rp
mov  0x8(%rax),%lp   add  $0x8,%ip
mov  -0x8(%ip),%rdx  mov  -0x8(%ip),%rdx
mov  %rdx,%rax       mov  %rdx,%rax
jmpq *%rax           jmpq *%rax
```

The sequence `unlocal ;s` is more efficient than `;s` jumping to `lp-trampoline` (see Section 3), especially if we combine the sequence into a superinstruction `unlocal-;s`. So, if the current definition contains locals, and if we know that `exit` returns from the current definition, we can compile `exit` into `unlocal ;s` as an optimization (through the intelligent `compile,`). If the definition performs return-address manipulation (so that the `exit` may return from a different definition), it first has to clean up the locals with `unlocal`. So, if the word contains `unlocal`, we disable this optimization.

2.2 does>

In addition to colon definitions, words defined with `does>` also call code that may define locals. We will use the following running example:

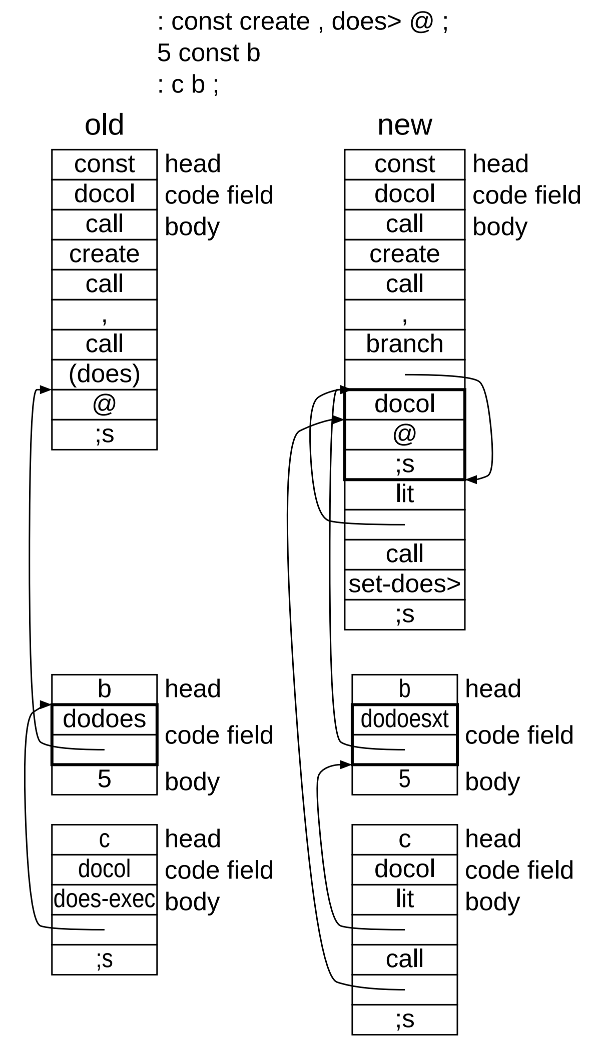

Figure 2: Old and new implementation of `does>`

```
: const create , does> ( A ) @ ;
5 const B
```

When running B, the code A after `does>` is called with either the primitive `does-exec` (when B is `compile,`d) or with the code-field routine `dodoes` (when B is `execute`d).

Now the `does` part may also define locals and contain `exit`, so we have to push the additional stuff on the return stack in these cases, too. Our first idea was to add `dodoesloc` and `does-exec-loc`, but that would have resulted in complications, so we soon came up with the following, better idea (see Fig. 2):

The code after `does>` (A in our example) is a full-blown colon definition with its own code field and execution token. Instead of using `dodoes`, which (after pushing the body address of B) calls the code at A as `call` does, we have `dodoesxt`, which `execute`s the xt of A. Here is the code for `dodoes`, compared to `dodoesxt` followed by `docol`:

```
dodoes                dodoesxt, docol
mov  %tos,(%sp)       mov  %tos,(%sp)
lea  0x10(%cfa),%tos  lea  0x10(%cfa),%tos
mov  %ip,%rdx
mov  0x8(%cfa),%ip    mov  0x8(%cfa),%cfa
sub  $0x8,%sp         sub  $0x8,%sp
                      mov  (%cfa),%rdx
                      mov  %rdx,%rax
                      jmpq *%rax
mov  %rp,%rax         mov  %rp,%rax
                      mov  %ip,%rdx
lea  -0x8(%rp),%rp    lea  -0x8(%rp),%rp
                      lea  0x18(%cfa),%ip
mov  %rdx,-0x8(%rax)  mov  %rdx,-0x8(%rax)
add  $0x8,%ip
mov  -0x8(%ip),%rdx   mov  -0x8(%ip),%rdx
mov  %rdx,%rax        mov  %rdx,%rax
jmpq *%rax            jmpq *%rax
```

The advantage for our locals problem is that no additional work is needed: the first locals definition in A changes the A colon definition into a `docolloc` colon definition, and there is no need to change the `dodoesxt`.

Again, `dodoesxt` is only used when B is `execute`d or called through a deferred word. When we `compile,` B, the intelligent `compile,` compiles the body address of B as literal, followed by `compile,`ing the xt of A, resulting in `call` or `call-loc` followed by the body address of A. We have added static superinstructions for `lit call` and `lit call-loc` to eliminate the overhead of executing two primitives instead of one. Here is the code for `does-exec` compared to that for the `lit-call` superinstruction:

```
does-exec             lit-call
mov  %tos,(%sp)       mov  %tos,(%sp)
                      mov  %ip,%rax
mov  (%ip),%tos       mov  (%ip),%tos
mov  %ip,%rdx         mov  0x10(%ip),%ip
mov  %rp,%rax         mov  %rp,%rdx
add  $0x8,%rdx        add  $0x18,%rax
lea  -0x8(%rp),%rp    lea  -0x8(%rp),%rp
sub  $0x8,%sp         sub  $0x8,%sp
mov  0x8(%tos),%ip
mov  %rdx,-0x8(%rax)  mov  %rax,-0x8(%rdx)
add  $0x10,%tos
add  $0x8,%ip         add  $0x8,%ip
mov  -0x8(%ip),%rdx   mov  -0x8(%ip),%rdx
mov  %rdx,%rax        mov  %rdx,%rax
jmpq *%rax            jmpq *%rax
```

Given that our `does>` is now based on taking an xt, we can make another interface to this functionality available: `set-does> ( xt -- )` changes the last defined word to first push its body address, then execute the xt. There are two benefits to `set-does>`:

First, when there is only one word between `does>` and `;`, one can pass that word (instead of a colon definition containing just that word) to `set-does>`, saving one call-exit pair at run-time. E.g.:

```
: const create , ['] @ set-does> ;
5 const B
```

When compiling B, this produces `lit @` (without additional effort), and saves a `call` and `;s` around the `@` at run-time.

The other advantage is that `set-does>` can be used more flexibly than `does>`, e.g., inside control structures; e.g, `struct.fs` contains

```
: dofield ( -- )
does> ( name execution: addr1 -- addr2 )
    @ + ;

: dozerofield ( -- )
    immediate
does> ( name execution: -- )
    drop ;

: field ( align1 off1 align size "name"
        --  align2 offset2 )
    2 pick >r create-field r> if \ off1<>0
        dofield
    else
        dozerofield
    then ;
```

In the usual case, a field should perform the `does>` part of `dofield`, but if the field has offset 0, then it should not compile anything, so it is defined as immediate word that does nothing (not even `0 +`, to avoid stack underflow at compile time). The properties of `does>` force this factoring, which I don't consider particularly conducive to understanding. With `set-does>`, we can define this as

```
: field ( align1 off1 align size "name"
        --  align2 offset2 )
    2 pick >r create-field r> if \ off1<>0
        [: @ + ;] set-does>
    else
        [: ;] set-does> immediate
    then ;
```

Another use case of `set-does>` is optimization:

```
: const ( n "name" -- )
  \ you must not change the body of "name"
  create ,
  ['] @ set-does>
  [: >body @ postpone literal ;] set-opt ;
```

`set-opt ( xt -- )` sets what happens when the created word is `compile,`d (it is the basis for the intelligent `compile,`). In this case it optimizes the word such that, instead of looking up the value at run-time, the lookup happens at `compile,` time and the resulting value is compiled as literal.

This can only happen after `does>` or `set-does>`, because `does>` and `set-does>` change what the word does, and that also overwrites any earlier `set-opt`. It is possible to implement the above with `does>`, but, like the `field` example, it would be more cumbersome.

Continuing onwards from `set-does>`, we could also define a defining word `does-create` `( xt -- )` that combines the functions of `create` and `set-does`, used as follows:

```
: const ( n "name" -- )
  ['] @ does-create , ;
```

The advantage of `does-create` would be that the defined word gets the final code field right from the start, instead of being first created with a `dovar` code field, and later overwritten with `dodoes` and (in our example) A; the current two-step approach leads to problems on Forth systems compiling to flash memory; while Forth implementors have found workarounds for these problems, it's better to provide an interface that does not need such workarounds. `Does-create` is not (yet?) implemented in Gforth.

Note that, while the new `does>` implementation makes the new locals-cleanup implementation simpler, the reverse is not true: You can do the new `does>` implementation (and `set-does>` and `does-create`) just fine in combination with the old style of locals-cleanup implementation.

3 Performance Impact

For a realistic evaluation of performance we would need a number of application benchmarks that spend a lot of time calling to and returning from definitions containing locals, and application benchmarks performing lots of calls to `does>`-defined words.

Unfortunately, we are not aware of benchmarks with these characteristics, so we use microbenchmarks here to evaluate the performance. Real applications may see much smaller performance differences than we see in these microbenchmarks.

Fig. 3 shows the results, and they will be explained in the following.

We call ten different words, with results from the old implementation shown in reddish colours and new implementations in bluish colours:

baseline An empty colon definition (without locals) that gives us a baseline.

0-locals A colon definition that is empty except that it contains the overhead of cleaning up locals (plus, for the new implementation, putting the additional stuff on the return stack). We measure three ways to clean up the locals: *old* is the old way of cleaning up locals (using `lp+!#`); *lp-trampoline* is the new implementation without using `unlocal`, so `;s` jumps to `lp-trampoline`; *unlocal* is the optimized variant of the new implementation that performs `unlocal` before `;s`, thus skipping `lp-trampoline`. Both new versions incur the overhead of pushing the additional data on the return stack with `call-loc` or `docolloc`.

3x0-locals If there are several words with locals, our new implementation (without `unlocal`) calls the same instance of `lp-trampoline` from each of these words, and the NEXT inside `lp-trampoline` then jumps to different code; this can lead to mispredicting this branch (depending on the indirect branch predictor of the CPU). 0-locals just has one such word and should not have problems with the branch predictor. For contrast, we also have 3x0-locals, where we have three instances of a word like the one used in 0-locals; for the *lp-trampoline* variant, this leads to a mispredicted NEXT in `lp-trampoline` on CPUs that use a branch target buffer (BTB) for predicting indirect branches. The number of calls and the number of loops is the same, so there should be no other differences from 0-local (except for the `execute` variants, where there is more overhead for handling three xts instead of one, and additional mispredictions, see below).

does A `does>`-defined word that just drops the address that `dodoes` (or its replacement) pushes on the stack. There are no locals in this set of words (the new `does>` implementation can also be implemented without changing the locals, and the performance should be independent). Here we also have three variants: the *old* one using `does-exec` and `dodoes`; the *new* one generating `lit call` (as a superinstruction) and `dodoesxt`; and finally a variant defined with `['] drop set-does>` that saves the call and return overhead.

We call these words in a loop in two ways: We `compile,` them into the loop, or we call them in a loop with `dup execute` (a little more complicated for the three-copy-variant). We also measure an empty loop and subtract its instructions, cycles, and branch mispredictions from the results to get an approximation of the pure cost of executing just that one word.

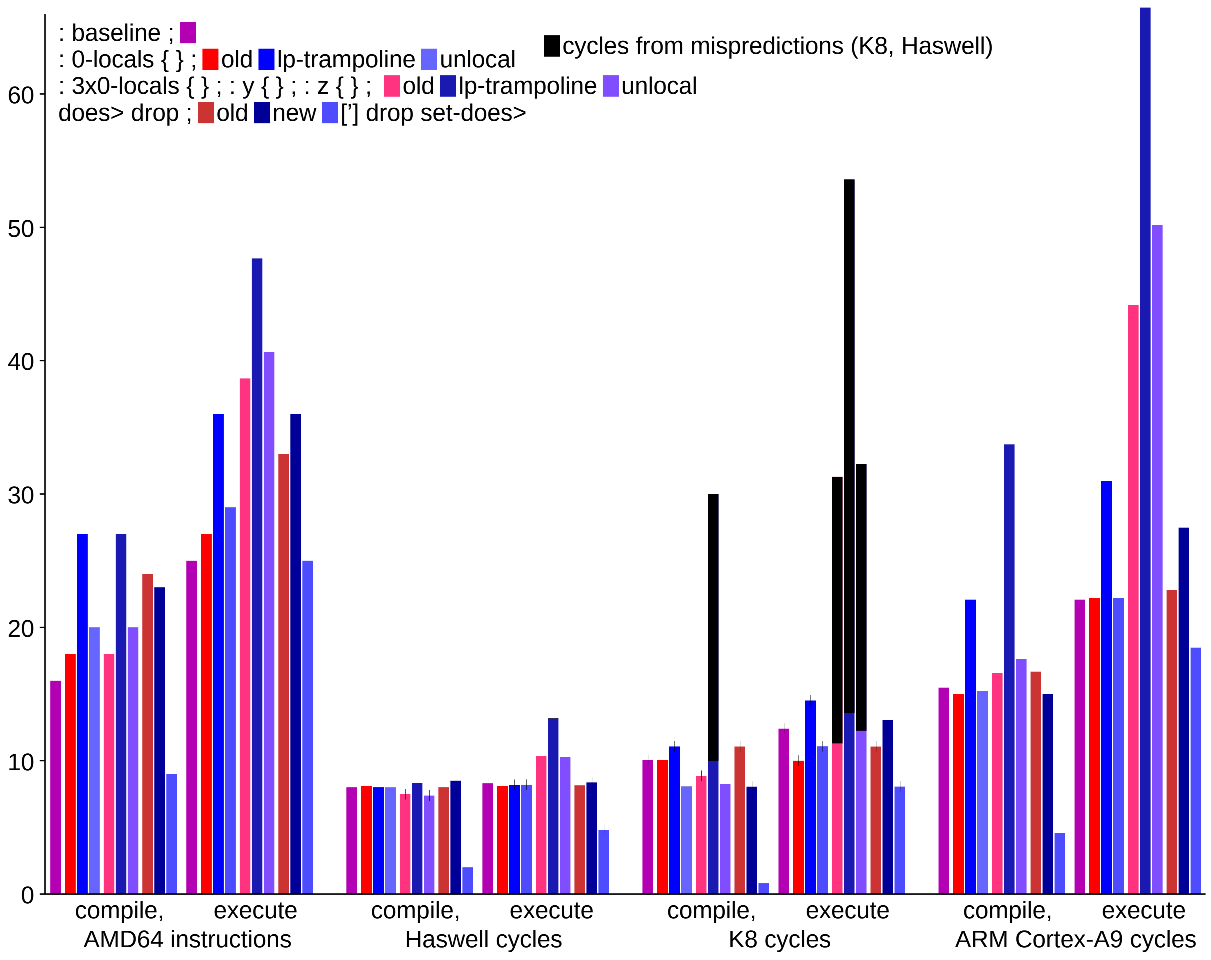

Figure 3: Instructions and cycles for performing a word (one invocation)

We used three different machines in our experiments: A Core i7-4790K (Haswell) based machine, an older (2005) Athlon 64 X2 4400+ (K8), and an ARM Cortex-A9 based PandaBoard ES. All machines ran Linux. We used the same binaries for the two machines with AMD64 architecture (Haswell, K8). On the Haswell and K8 we measured instructions, cycles, and branch mispredictions using performance counters; on the Cortex-A9 we measured CPU time with `time`, and computed cycles from that.

3.1 Locals performance

The first set of columns shows the AMD64 instructions executed when running the words. We see that, in the old implementation, cleaning up the locals stack takes two additional instructions over the locals-less baseline, and the *lp-trampoline* implementation takes 9 more instructions than the old one; the *unlocal* implementation costs only 2 instructions more than the old implementation. The same differences are seen in both the `compile,`d variant and in the `execute`d variant, but the baseline is higher for the `execute`d variant.

These differences in instruction count are not reflected in the Haswell cycles; this processor apparently manages to execute most of the additional instructions in parallel to the instructions that it already performs in the baseline. But why can it not extract more parallelism from the baseline? There is probably a data-dependence chain having to do with the Forth VM instruction pointer (ip).[3] In more realistic code there is more code inside the loops and definitions, so ip-based dependency chains probably do not usually determine performance in realistic code. Therefore, the instructions counts may be a better indicator of the performance impact of our changes on real code than the Haswell cycle counts.

The Haswell has a very good branch predictor [RSS15], so branch mispredictions don't play a significant role on Haswell, even for 3x0-locals.

The K8 also mostly shows few performance differences between the implementations of the locals,

[3]Save ip to the return stack, load it back, load the target of the `(loop)` primitive, and perform a few additions in between.

except that there are big differences in some cases coming from branch mispredictions (the K8 predicts indirect branches with a BTB). We estimate 20 cycles penalty per misprediction, and have coloured the corresponding part of the bars in black; comparing the non-black part of the 3x0-locals bars with the 0-locals bars, this estimate is about right. The `compile,`d 3x0-locals benchmark causes one misprediction with the *lp-trampoline* variant, from `lp-trampoline`, as discussed above. Optimizing the new locals implementation with `unlocal` eliminated this slowdown.

The `execute`d 3x0-locals benchmark has an additional branch misprediction, in `docol/docolloc`, with all implementations.

The ARM Cortex-A9 timing results seem to be influenced by instructions counts (which are probably be similar to the AMD64 counts), and (comparing 0-locals with 3x0-locals) also by mispredictions in a way similar to the K8 results, so the Cortex-A9 probably also has a BTB. Unfortunately, we do not have performance counter results for this CPU, so we cannot present misprediction results (nor instruction counts).

Concerning the difference between the old and the new locals cleanup implementation, we see that, on the Cortex-A9, *lp-trampoline* is quite a bit of slower than the old implementation, but the `unlocal` implementation has similar performance as the old implementation.

3.2 `DOES>` performance

When `compile,`d, the new implementation of the `does>`-defined word uses one instruction less (with the `lit call` superinstruction) than the old implementation. There are also corresponding small differences in the cycles on the K8 and Cortex-A9; on the Haswell the new implementation takes 0.5 cycles more than the old one.

When `execute`d, the new implementation takes three additional instructions; on the K8 and Cortex-A9 this is also reflected in the number of cycles, while there is little difference on the Haswell.

The `['] drop set-does>` variant saves 15 instructions for the `compile,`d version and 8 instructions for the `execute`d one compared to the old implementation. It also gives good speedups on all CPUs; this time this even includes the Haswell, because this variant shortens the dependence chain.

4 Native-code Caveats

If implemented naïvely, the additional return address can have a high cost on native-code systems that (unlike Gforth) use the architecture's return instruction for implementing `exit`. Return instructions on modern CPUs have a special branch predictor that is called *return stack* (yes, the same name as the Forth return stack, and it also contains return addresses, but it's not programmer-visible). A return to the address of the corresponding call normally predicts correctly, and a return to a different address causes a misprediction (about 20 cycles penalty on a modern CPU). Therefore each return address should be produced by a call instruction, and not manipulated. One way to achieve this in a native-code system is to push the additional (Forth) return stack items as follows:

```
push data needed for cleaning up locals
call rest-of-definition
clean up locals
ret
rest-of-definition:
...
ret
```

5 Conclusion

If we require that `['] exit execute` works, we have to clean up locals in a compatible way. The popular technique of pushing extra data and the return address of a cleanup code fragment on the stack works, but has some performance caveats. Fortunately, we achieve performance similar to the old implementation in most cases by optimizing `exit` to perform `unlocal ;s`.

The new locals cleanup implementation also led to a new `does>` implementation (but the new `does>` implementation can be implemented without the new locals cleanup). The performance for code using `does>` is comparable to the old implementation; but the new implementation also makes it possible to use `set-does>`, which allows more flexibility in structuring words, and may save a call-return pair, increasing performance.

References

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

[RSS15] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch prediction and the performance of interpreters — don't trust folklore. In *Code Generation and Optimization (CGO)*, 2015.