40th EuroForth Conference

Newcastle, England
September 27-29, 2024

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 40th EuroForth finds us in Newcastle upon Tyne; in 2023 EuroForth was in Rome, and in 2022 it was online. Information on earlier conferences can be found at the EuroForth home page (`http://www.euroforth.org/`).

Since 1994, EuroForth has a refereed and a non-refereed track. This year there have been two submissions to the refereed track, of which one was accepted (50% acceptance rate). For more meaningful statistics, I include the numbers since 2006: 32 submissions, 23 accepts, 72% acceptance rate. The reviews of all papers are anonymous to the author: All papers were reviewed and the final decision taken without involving the author, including the submission coauthored by a program-committee member. I thank the program committee for their paper reviews and the authors for their submissions.

Several papers were submitted to the non-refereed track in time to be included in the printed proceedings. Late papers as well as slides and links to videos will be included in the final proceedings (`http://www.euroforth.org/ef24/papers/`). I thank the authors for their papers. In addition to the papers and presentation handouts available before the conference, these online proceedings also contain papers and presentation handouts that were provided at or after the conference. Also, some of the papers included in the printed proceedings were updated for these online proceedings. I thank the authors for their papers and slide handouts.

You can find these proceedings, as well as the individual papers and slides, and links to the presentation videos on `http://www.euroforth.org/ef24/papers/`.

Workshops and social events complement the program. This year's EuroForth is organized by Bill Stoddart and Janet Nelson.

Anton Ertl

Program committee

M. Anton Ertl, TU Wien (chair)
Marcel Hendrix, Eindhoven University of Technology
Ulrich Hoffmann, FH Wedel University of Applied Sciences
Matthias Koch
Jaanus Pöial, Tallinn University of Technology
Bradford Rodriguez, T-Recursive Technology
Bill Stoddart
Reuben Thomas

Contents

Towards a Prospective Values semantics for a reversible Forth

Bill Stoddart, Frank Zeyda

September 5, 2024

Abstract

We describe a "prospective values" semantics for Forth, including the backtracking extension provided by our reversible virtual machine RVM-Forth. We use S $\diamond$ E to represent the value expression E would have were it to be evaluated after the execution of program S. We call this the prospective value of E after S. This form is expressive enough to describe the semantics of an extended form of guarded command language that incorporates backtracking and speculative computations. We give semantics for Forth stack commands, assignments, speculative computations, conditionals and loops. We sketch the work that remains to be done.

1 Introduction

We write S $\diamond$ E for the value expression E would have were it to be evaluated after the execution of program S. We call this the prospective value of E after S. In this paper, where we apply this idea to Forth, S is a Forth program, i.e. some self contained Forth code, and E is a mathematical expression.

For example x 1 + to x $\diamond$ $10 * x \;=\; 10 * (x + 1)$

Note the large equals $=$ is a very low priority equals symbol. The symbol $\diamond$ is next lowest in priority.

We have developed the theory of prospective value semantics (PV semantics) in a series of papers, most recently in [DFM$^+$23]. The presentation is via a guarded command language bGSL (backtracking generalised substitution language), and provides a formalism for describing backtracking and reversible computations. Our theory is developed as an extension of the B-Method [Abr96], and uses an unconventional version of set theory proposed by Eric Hehner [Heh93, Heh23]. The integration of Hehner's ideas into the set theory of B requires a new theory with its own logic, which we describe in [SDMZ24]. We have developed a reversible Forth [SZL10] to act as an implementation platform, and developed some compilation techniques that make special use of Forth's essential features, for example executing type tagged parse trees as Forth programs [RS10].

The limited semantics we present here has the same level of abstraction as our guarded command language bGSL. We do not deal with CREATE..DOES>, or with direct access to memory locations. We deal with "values" which are held on the stack after being obtained from obtained from named variables and data structures or calculated from other stack values, and which may be assigned to such variables and data structures. On the other hand the language we describe is much more expressive than a traditional guarded command language in that we describe program structures for backtracking and speculative computations. Indeed, a principle motivation for providing a PV semantics for RVM-Forth is to provide a means of checking the validity of the code produced by such a compiler for our backtracking guarded command language bGSL which uses our reversible Forth as its target language.

When constructing a PV semantics for Forth we have to take into account the following:

- The way Forth expressions are written, in an extended postscript notation with explicit stack manipulations, is so different from how expressions are written in mathematics that we will have to abandon the convenient and unspoken fiction that program expressions and mathematical expressions are one and the same.
- Forth has an explicit stack so we need a way to represent the stack as a mathematical expression,

2 The stack, part 1

Our semantics uses the typed set theory of B. A stack may hold items of different type, and in a typed theory this prevents us from representing it as a sequence. However, we can represent a stack containing different types of value as a tuple [1].

We use the symbol ε to represent the empty parameter stack, and in our mathematical universe we give the parameter stack the name s.

Here are some examples showing the value taken by the stack following some simple Forth code. The operation SP! clears the stack to give us a defined starting state.

SP! $\diamond\ s \ = \ \varepsilon$
SP! 1 $\diamond\ s \ = \ \varepsilon \mapsto 1$
SP! 1 2 $\diamond\ s \ = \ \varepsilon \mapsto 1 \mapsto 2$
SP! 1 2 10 $\diamond\ s \ = \ \varepsilon \mapsto 1 \mapsto 2 \mapsto 10$
SP! 1 2 10 + $\diamond\, s \ = \ \varepsilon \mapsto 1 \mapsto 12$

Since the stack always consists of a tuple that commences with ε we can take the liberty of omitting the ε when the stack is non-empty and replacing the

[1] This means the type of the stack will change every time we push or pop a value. Thus the stack has no identifiable type, but every state of the stack does have a type.

maplet symbol $\mapsto$ by a space. This allows the above results to be expressed as follows.

SP! $\diamond$ s $=$ ε
SP! 1 $\diamond$ s $=$ 1
SP! 1 2 $\diamond$ s $=$ 1 2
SP! 1 2 10 $\diamond$ s $=$ 1 2 10
SP! 1 2 10 + $\diamond$ s $=$ 1 12

3 Expressions and the semantics of assignment

Let E be Forth code that's only effect is to leave one item on the stack, i.e. it causes no change of state of program variables or any memory; we will call such a fragment of code an "expression". We will want to use the value left by E in some of our semantic equations.

In the form S $\diamond$ E, The text to the left of the diamond is Forth code, and that to the right is a mathematical expression, i.e. the diamond separates Forth code from mathematical text, and we need a notation to translate the Forth expression E into the mathematical world. We enclose E in semantic brackets $[\![$E$]\!]$ to represent this translation. Some examples will make this clear.

Suppose x is a Forth VALUE holding an integer. We translate x into the mathematical value x

$[\![$x$]\!] = x$

Next consider the translation of a simple expression:

$[\![$x 10 +$]\!] = x + 10$

In the next example we use a symbolic stack trace to perform the translation $[\![$x DUP DUP $*$ +$]\!] = x^2 + x$

Forth commands	Stack
x	x
DUP DUP	$x\ x\ x$
$*$	$x\ x^2$
+	$x^2 + x$

Here is an symbolic trace for an example where three arguments a b c are provided from the stack:

$[\![$ a b c -- 2DUP * -ROT + SWAP -ROT * + 2* $]\!]$

Forth commands	Stack
a b c -- 2DUP	$a\ b\ c\ b\ c$
*	$a\ b\ c\ b{*}c$
−ROT	$a\ b{*}c\ b\ c$
+	$a\ b{*}c\ b{+}c$
SWAP	$a\ b{+}c\ b{*}c$
−ROT	$b{*}c\ a\ b{+}c$
*	$b{*}c\ a{*}(b+c)$
+	$b{*}c + a{*}(b+c) \ = \ a{*}b + a{*}c + b{*}c$
2*	$2*(a*b + a*c + b*c) \ = \ 2*ab + 2*ac + 2*bc$

4 Assignment

To avoid continual use of the semantic brackets $[\![..]\!]$ we will use a change in typeface, by which the Forth expression E is translated as the mathematical expression E.

The semantics of changing variable states is expressed in lambda notation. $(\lambda\, x \bullet F)E$ represents the rewriting of F with with the term E substituted for each occurrence of x in F. For example $(\lambda\, x \bullet 2*x)(y+10) \ = \ 2*(y+10)$.

In Forth, and with E an expression as defined above (i.e. Forth code that leaves a value on the parameter stack and causes no other change of state) E to x represent the assignment of the value left by E to the Forth VALUE x. We can give its semantics by describing its effect on a general expression F:

E to x $\diamond$ $F \ = \ (\lambda\, x \bullet F)E$

For example:

$x\ 10\ +\ to\ x\ \diamond\ 2*x+y \ =$ by rule for assignment
$(\lambda\, x.2*x+y)[\![x\ 10\ +]\!] \ =$ by semantics of expression
$(\lambda\, x.2*x+y)(x+10) \ =$ by lambda evaluation
$2*(x+10)+y$

5 The stack, part 2

We represent the stack mathematically as a tuples, so let us review tuple notation. We write $x \mapsto y$ for the tuple consisting of the pair of values x and y, $x \mapsto y \mapsto z$ for the tuple consisting of the triple of values x, y, and z. The tuple operator is a left associative binary operator, so $x \mapsto y \mapsto z \ = \ (x \mapsto y) \mapsto z$.

We can decompose a tuple into its first and second components with the functions L (left) and R (right).

In line with Forth usage in referring to the top and next from top elements of the stack we define the following functions.
$$top(s) \ \widehat{=} \ R(s)$$
$$next(s) \ \widehat{=} \ R(L(s))$$

Unlike an assignment to a VALUE, e.g. 3 to X, which changes the whole of

X, stack operations may affect only part of s. So our approach will be to use "helper functions" to describe the whole new stack, and assign this whole new state.

For following are examples of these helper functions:

$drop(s) \ \widehat{=} \ L(s)$
$twodrop(s) \ \widehat{=} \ L^2(s)$
$nip(s) \ \widehat{=} \ L^2(s) \ \mapsto \ R(s)$
$swap(s) \ \widehat{=} \ L^2(s) \mapsto top(s) \mapsto next(s)$
$plus(s) \ \widehat{=} \ L^2(s) \mapsto (next(s) + top(s))$
$minus(s) \ \widehat{=} \ L^2(s) \mapsto (next(s) - top(s))$

and so on

Then to describe the value of expression E after a stack operation OP we have

$\text{OP} \diamond E \ = \ (\lambda\, s.E)op(s)$

Where E is a stack expression, such as $L(s)$, or just s. An example in the next section should help to make this clear.

6 Sequential Composition

Our semantic rule for sequential composition is:

sequential composition $\quad \text{S T} \ \diamond \ E \ = \ \text{S} \diamond \text{T} \diamond E$

Note that $\diamond$ is right associative, so:

$\text{S} \diamond \text{T} \diamond E \ = \ \text{S} \diamond (\text{T} \diamond E)$

We can use this rule to show that the effect of NIP on the stack s is equivalent to that of SWAP DROP.

SWAP DROP $\diamond \ s \ =$ by rule for sequential composition
SWAP $\diamond$ DROP $\diamond \ s \ =$ by semantics of DROP
SWAP $\diamond \ (\lambda\, s \bullet s)drop(s) \ =$ by lambda evaluation
SWAP $\diamond \ drop(s) \ =$ by semantics of SWAP
$(\lambda\, s \bullet drop(s))swap(s) \ =$ by lambda evaluation
$drop(swap(s)) \ =$ applying *swap*
$drop(L^2(s) \mapsto top(s) \mapsto next(s)) \ =$ applying *drop*
$L^2(s) \mapsto top(s) \ =$ semantics of NIP
NIP $\diamond \ s$

It seems strange that we compute the effect of SWAP DROP on the stack by first computing the effect of DROP and *then* computing the effect of SWAP, but the intermediate result $drop(swap(s))$ in the above derivation shows the helper function for SWAP is applied before that of DROP in obtaining the result.

7 Guard, choice and backtracking

Let g be a condition test that leaves either a true flag or a false flag on the stack and has no other side effect. The construct g => is a guarded no-op. If g leaves a true flag, the flag is removes and execution continues ahead. If g leaves a false flag, the effect is to reverse computation. In this case there is no state after g. Mathematically. we represent this as *null*, where *null* represents nothing. In our mathematical semantics we capture the idea of nothing by using Eric Hehner's Bunch Theory [Heh93]; this is a reformulation of set theory in which the collection and packaging of elements are orthogonal activities. This gives us access to unpackaged collections. We use $\sim S$ to represent the unpacking of set S. For example $\sim\{1,2\} = 1,2$ where $1,2$ is an unpackaged collection. The comma in $1,2$ is now a mathematical operator, known as bunch union. We obtain *null* by unpacking the empty set.

$null = \sim\{\ \}$

Bunch union has the properties: $S,T = T,S$ and $S, null = S$, and an additional property of *null* is $\{\ null\ \} = \{\ \}$.

Corresponding to the programming guard => , we have a bunch guard $\twoheadrightarrow$ in our mathematical notation, defined by the following equations:

$$true \twoheadrightarrow E = E,\ \ false \twoheadrightarrow E = null$$

so the expression $x = 1 \twoheadrightarrow x$ has the value 1 if x=1, and is equal to *null* for any other value of x.

Evidently, this is a very unconventional mathematical theory, and when we began to use it we has some papers rejected by referees who were concerned that Hehner's bunch theory had never been formally demonstrated to be a valid theory, in that it had never been given a "model" (a translation that re-expressed it in terms of standard set theory). These concerns lessened after a model for a version of bunch theory was published [MB01], and we have given a model for our version of bunch theory in [SDMZ24].

Our semantic rule for guard is

g => $\diamond E = g \twoheadrightarrow E$

here g is the mathematical translation of the Forth guard g, which for any specific g we can represent more fully using our semantic brackets, e.g.

$[\![\ \text{x 1 =}\ ]\!] = x = 1$

Guards combined with choice can describe control structures, including backtracking.

We introduce a Forth choice operation. S_1 [] S_2 presents a choice between executing S_1 or S_2. This choice has to be bracketed, rather like an IF construct, as

<CHOICE S_1 [] S_2 [] ... CHOICE>

The semantic rule for choice is:

S [] T $\diamond$ E $=$ $(S \diamond E)$, $(T \diamond E)$

Here the comma on the RHS is the bunch union operator that we defined above. the rule does not say which choice is tried first.

For example:

<CHOICE 1 to x [] 2 to x CHOICE> $\diamond$ x $=$ $1, 2$

The combination of choice and guard allows us to express backtracking. Consider:

<CHOICE 1 to x [] 2 to x CHOICE> x 2 = => $\diamond$ x

This has the following operational interpretation: a choice is made to assign either 1 of 2 to x, then a guard checks if x=2. If it does computation continues ahead, otherwise we backtrack to the previous choice and continues ahead once more with the unused choice being selected. This time x *will* be set to 2 and the guard lets computation continue ahead. This simple example shows how we can use a guard to *retrospectively* select from two choices.

The semantic analysis goes as follows:

<CHOICE 1 to x [] 2 to x CHOICE> x 2 = => $\diamond$ x $=$ by semantics of sequential composition

<CHOICE 1 to x [] 2 to x CHOICE> $\diamond$ x 2 = => $\diamond$ x $=$ by semantics of program guard

<CHOICE 1 to x [] 2 to x CHOICE> $\diamond$ $x = 2 \rightarrowtriangle x$ $=$ by semantics of choice

1 to x $\diamond$ $x = 2 \rightarrowtriangle x$, 2 to x $\diamond$ $x = 2 \rightarrowtriangle x$ $=$ by semantics of assignment

$1 = 2 \rightarrowtriangle 1$, $2 = 2 \rightarrowtriangle 2$ $=$ by property of bunch guard

$null, 2$ $=$ by property of null

2.

7.1 Conditionals

We can think of g IF S ELSE T THEN $\diamond$ E as a bunch union of two terms, corresponding to the two branches of the conditional, and with the term corresponding to the branch not taken being equal to *null*. In our semantics this is expressed as follows:

g IF S ELSE T THEN $\diamond$ E $=$ $(g \rightarrowtriangle \text{S} \diamond E)$, $(\neg\, g \rightarrowtriangle \text{T} \diamond E)$

Once again we note the change of typeface from g to g which represents the conversion of the program expression g , indicating code returning a flag, and the g in mathematical typeface, which represents the mathematical predicate corresponding to g.

8 Speculative computation

In our semantics S $\diamond$ E expresses the value E would take after executing S. We can use the same semantics to describe a speculative computation which executes S, evaluates and saves the result of E, then reverses, undoing any changes made in the forward execution of S. Thus we obtain, in our program, the value E would have after S but without incurring any of the side effects produced by executing S.

As with choice we needs brackets to express this:

<RUN S E RUN>

is a programming structure which adds to the stack the value E produces if executed after S, but without incurring the side effects that execution of S may produce. Its semantics is:

<RUN S E RUN> $\diamond s \;=\; s \mapsto$ (S $\diamond$ E)

If S contains choices there may be a plurality of values that E could take, and we can collect. If these are integer values the construct to do this is:

INT { <RUN S E RUN> }

In this case S $\diamond$ E will be a bunch, and we have the following semantic rule, in which, once again, s is our mathematical representation of Forth's parameter stack.

INT { <RUN S E RUN> } $\diamond s \;=\; s \;\mapsto\; \{$S $\diamond$ $E\}$

8.1 Example, Pythagorean triples, with a new concept of function application

We need to introduce some additional aspects of the RVM sets package.

The mathematical notation $m..n$ where $n \geq m$, represents the set of numbers $\{m,\ m+1,\ ...\ n\}$ We provide this as a postfix operator in RVM Forth, used as, e.g.

1 4 .. .SET <cr> {1,2,3,4} ok

We have CHOICE from a set, used as in the following example. CHOICE makes a provisional choice form a set that may be revised by backtracking.

INT { <RUN 1 4 .. CHOICE 10 * RUN> } .SET <cr> {10,20,30,40} ok

We now consider a program to produce a set of Pythagorean triples $\{a, b, c\}$ where $a^2 + b^2 = c^2$. In the code we choose values for a and b, calculate a^2+b^2 and then apply a perfect square root function PERF. This function illustrates the "new concept of function application" we mentioned above. The idea of

n PERF

is that it returns the perfect square root of n, if that exists, or otherwise triggers

backtracking. In the following examples we see that if backtracking continues back to the user console, we get the prompt ko rather than ok.

```
0 PERF .<cr> 0 ok
1 PERF .<cr> 1 ok
2 PERF .<cr> ko
3 PERF .<cr> ko
4 PERF .<cr> 2 ok
```

This may seem a programming trick - we have just included the guard that triggers backtracking within the code for PERF. However, we have *mathematical* reason to claim that this is indeed a new idea of function application. Working with integers and using $\sqrt{n}$ to represent the perfect integer square root of n, it is clear for example that no integer satisfies $\sqrt{2}$, and we capture this in our theory by saying $\sqrt{2} = null$. We also recall that from the semantics of guards, it is a *null* result that triggers backtracking. The new concept of function application is that a function application might represent "nothing", which we cannot express without the *null* of bunch theory. To express the stack effect of PERF we need to specify that if the stack input parameter n has an integer square root m than that will be the stack output parameter, otherwise *there will be no stack after state*. To do this we use *null*, as follows.

PERF (n -- **if** $\exists\, m \bullet m^2 = n$ **then** m **else** $null$ **end**)

Now for the program to produce set of Pythagorean triples with perpendicular sides less than n. The set we are producing here is a set of sets of numbers, and its mathematical type is $\mathbb{P}(\mathbb{N})$. This is represented in our Forth sets package, in postfix, by the type signature INT POW. [2]

```
: TRIPLES ( n -- s, s is a set of Pythagorean triples with adjacent sides ≤ n )
   (: n :)
   INT POW {
      <RUN
         1 n .. CHOICE to A
         A n .. CHOICE to B
         A B COPRIME ->
         A DUP * B DUP * + PERF to C
         INT { A , B , C , }
      RUN>
   } ;
```

In this code, A, B and C are global VALUEs. The COPRIME guard prevents similar triangles being included, for example {3,4,5} and {6,8,10}.

Here is an example run

```
100 TRIPLES .SET <cr> {{3,4,5},{5,12,13},{7,24,25},{8,15,17},{9,40,41},
{11,60,61},{12,35,37},{13,84,85},{16,63,65},{20,21,29},{20,99,101},{28,45,53},
{33,56,65},{36,77,85},{39,80,89},{48,55,73},{60,91,109},{65,72,97}}ok
```

[2] Whilst Forth is untyped, our sets package only supports the typed sets allowed in Abrial's set theory.

9 Preconditions

In general, in the field of formal semantics, operations are taken to have specific conditions which render them safe for use. These "preconditions" are there to protect us attempting to access the 20th element of a 10 element array, taking the square root of a negative number, dividing by zero etc. Unlike a guard, a pre-condition does not control whether an operation can take place, rather it is part of the instructions of using the operation. In Forth the situation with respect to pre-conditions is complex, because the programmer takes responsibility for an operation being meaningful in a particular context. For example, in 32 bit arithmetic, 7FFFFFFF 1 + violates a precondition of + if we are using signed arithmetic, but not for unsigned arithmetic. However, one universal precondition of + is that it requires at least two elements to be on the stack.

We use the symbol $\perp$ to express the effect of violating a precondition. The idea is that $\perp$ represents absolute unpredictability - more unpredictable than just allowing any possible result - there might be no result because the computation does not terminate, or the machine might blow up!

We use P | S to represent P as the pre-condition for S. Our rule for preconditions is:

$\mathrm{P} \mid \mathrm{S} \diamond E \;=\; (P \;\rightarrowtriangle\; S \diamond E)\;,\; (\neg\, P \;\rightarrowtriangle\; \perp)$

We interpret $\perp$ as a maximally non-deterministic bunch. We can think of the unpackaged collections of bunch theory as representing non-determinism or uncertainty, e.g. the bunch 1,2 representing a value that might be 1 or might be 2. In this knowledge based order the value $\perp$ represents a value about which nothing can be known, not even whether it exists, and *null* can be taken as the object about which too much is known, to the point of contradicting its possible existence. It is at the other end of the scale from $\perp$.

10 Loops

We consider the treatment of a WHILE loop

BEGIN g WHILE S REPEAT

Here g is some Forth code which leaves a flag in the stack and otherwise leaves the program state unchanged.

Following the B-Method (and adapting it to Forth) the programmer is required to provide formal comments which identify a an invariant expression I and a variant expression V for the loop.

The invariant expression must have the property

$\mathrm{S} \diamond E \;=\; E$

When the loop terminates the invariant expression will still have the same value, but the condition reported by g is false. This allows us to draw a conclusion

about the effect of the loop.

The variant expression serves the purpose of ensuring that the loop *does* terminate. It has to be an expression that is greater than 0 and decreased by S. Obviously this cannot continue for ever, so the existence of such an expression implies that the loop must terminate. Its formal property is:

$V > 0 \wedge (\mathrm{S} \diamond V) < V$

We illustrate this method using Euclid's algorithm for the calculation of the greatest common divisor of two numbers.

```
: GCD ( a b – c, a>0 ∧ b>0 | c = gcd(a,b) )
   BEGIN (
   INVARIANT gcd(top(s), next(s))
   VARIANT top(s) + next(s) )
      2DUP ≠
   WHILE
      2DUP > IF SWAP THEN
      OVER -
   REPEAT DROP ;
```

First note that we have a pre-condition that requires a>0 and b>0. Since a, b are names for the top two stack elements, this ensures that our variant property $V > 0$ holds. Depending on the branch taken by the IF, we have S $\diamond$ V = $top(s)$ or S $\diamond$ V = $next(s)$, and since V = $top(s) + next(s)$ in both cases we have S $\diamond$ V < V. So the variant properties are satisfied and we can be sure the loop terminates.

That the loop invariant holds follows from the mathematical property $y > x \Rightarrow gcd(x, y) = gcd(x, y - x)$. When the loop terminates the loop condition tells us that $next(s) = top(s)$ and the loop invariant tells us $gcd(top(s), next(s) = gcd(a, b)$ Thus we have two copies of the required result on the stack and just have to drop one of them to complete the computation.

11 Pointers and immutable objects

We identify immutable objects with their pointers. This allows us to treat such pointers as values.

Consider this floating point interaction in `RVM_FORTH`.

```
2. DUP 1. F F. F. <cr> 3 2 ok+
```

A floating point literal creates the floating point value in memory and leaves a pointer to that value on the parameter stack. Floating point operations work via these pointers.

Such an approach helps us formulate our semantics but entails a need for garbage collection: in the above interaction garbage values 1. 2. and 3. are left in memory when the computation is complete. We address this by collecting garbage during reverse computation, an idea first proposed by Henry Baker [Bak92].

12 Future work

Our account of prospective values in Forth is far from complete.

One area we have not discussed is the semantics of pointers which represent mutable objects. We recall that an array, for example, is implemented as a mutable object because we want the ability to individually modify one of its elements without creating a new instances of the whole array. Our semantics regards a reference as equivalent to the object referred to, and thus has no way of distinguishing the duplication of a reference to an object from the duplication of the object itself. Duplicating a reference to a mutable object invalidates our semantics, and we need to investigate restrictions, or "healthiness conditions" [HJ98] that ensure this does not happen in a given program text.

One reason we need references to mutable objects is to pass them as parameters. However, we recall that passing an array by reference is not the same as passing it as a value, because, when it is passed as a reference changes to the array occur on the original array, whereas changes to value parameters take place in the stack frame, which is discarded on exit. To keep our semantics consistent we must limit assignments to immutable reference parameters, and treat the passing of such parameters using call by name [B$^+$60]. Call by name can reserve surprises for the unwary, but Abrial [Abr96] has given a restricted call by name semantics for B.

It also remains to define the semantics of parameter passing in the presence of an explicit stack, to investigate how this may be affected by how local variables are implemented, and see whether we can define a semantics of parameter passing that is valid for diverse implementations.

It will be interesting to explore which problems can and cannot be efficiently solved using the reversible programming structures presented here, and to present example case studies. In [SDMZ24] we use a chess puzzle, the circular knight's tour, to show the power of prospective value calculations: these are used to implement a heuristic that chooses moves to the most tightly constrained squares. This is presented in our reversible guarded command language bGSL [DFM$^+$23], and needs to be complemented by case studies in our reversible Forth.

13 Conclusions

We have shown how prospective value semantics provides a description of stack based operations, speculative computations and backtracking, but a full description of Forth semantics, covering interpretation and compilation, memory access, and the definition of defining words, is beyond the scope of the theory presented here, even when it is extended as outlined under future work.

When transporting prospective value semantics from our usual B like environment to Forth, the extended postfix used in Forth forces us to distinguish more clearly between programming and mathematical notations. Forth has a finer grained semantics, where an expression is defined as as a sequence of opera-

tions, rather than in the mathematical notation of an expression sub-language. This additional detail can be captured in two ways by the semantics we investigate here. Either we can translate postfix expressions to infix in order to describe their effect (and this might require us to write our Forth in a particular way, and might be particularly useful in analysing the output of a compiler for our backtracking guarded command language bGSL [DFM$^+$23]), or we can process them at the level of the individual Forth operations of which they are comprised. In both cases we can include the effect of stack manipulations in our analysis.

In formulating a semantics of prospective values, the mathematical expression of nothing as the constant *null* plays the key role of representing expression values arising from program branches that are not taken. We also use it to illustrate a new form of function application, in which a mathematical function application can yield *null* to indicate that the described object does not exist, with the matching operational interpretation being that such an application triggers backtracking.

References

[Abr96] J-R Abrial. *The B Book*. Cambridge University Press, 1996.

[B$^+$60] J. W. Backus et al. Report on the algorithmic language Algol60. *Communications of the ACM*, 3.5, 1960.

[Bak92] H G Baker. The Thermodynamics of Garbage Collection. In Y Bekkers and Cohen J, editors, *Memory Management: Proc IWMM'92*, number 637 in LNCS, 1992.

[DFM$^+$23] S E Dunne, J F Ferreira, A Mendes, C Ritchie, W J Stoddart, and F Zeyda. bGSL: An imperative language for specification and refinement of backtracking programs. *Journal of Logical and Algebraic Methods in Programming*, 130, 2023.

[Heh93] E C R Hehner. *A Practical Theory of Programming*. Springer Verlag, 1993.

[Heh23] E C R Hehner. *A Practical Theory of Programming, 2023 edition*. University of Toronto, 2023. Available at https://www.cs.toronto.edu/ hehner/aPToP/aPToP.pdf.

[HJ98] C A R Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.

[MB01] J Morris and A Bunkenburg. A theory of bunches. *Acta Informatica*, 37(8), 2001.

[RS10] C Ritchie and W. J. Stoddart. A Compiler which Creates Tagged Parse Trees and Executes them as FORTH Programs. In *26th EuroForth Conference Proceedings*, 2010.

[SDMZ24] W J Stoddart, S E Dunne, C Mu, and F Zeyda. Bunch theory: axioms, logic, applications and model. *Journal of Logical and Algebraic Methods in Programming*, 140, 2024.

[SZL10] W J Stoddart, F Zeyda, and A R Lynas. A virtual machine for supporting reversible probabilistic guarded command languages. *ENTCS*, 253, 2010.

Using a container to provide 32 bit gcc5 compilation services on a 64bit Linux system

Bill Stoddart

June 30, 2024

Abstract

At EuroForth 2023 we mentioned that our reversible Forth had become impossible to maintain because the 32 bit gcc compiler is no longer supported. It was suggested that "containers" might provide a solution. Here we report on where that suggestion has led us.

1 Introduction

According to Red Hat, "A Linux container is a set of processes that are isolated from the rest of the system. All the files necessary to run them are provided from a distinct image."

This definition is fine so long as we don't take the term "isolation" too seriously. We will be using a 32 bit linux running in a container to provide 32 bit gcc compilation services required by the 64 bit Linux host system. The source files to be compiled and the resulting binary files will be on the 64 bit host, but all compilation will be done within the container. This will be possible because we are able to mount a directory of the host into the file system of the Linux running in the container. Apart from this the processes in the container are isolated from the host system.

Specific technologies used are the Docker package for containers, and 64 bit Fedorer 38 as the host. The name of our reversible Forth is `RVM_FORTH`.

2 Overview

2.1 Building our container image

We did not have to create a container from scratch. Docker provides an archive of pre-built container images, along with a search facility, and in the repository we found the entry:

```
frankwolf/32bit-ubuntu                32bit Ubuntu docker image
```

We pulled down this image from the reposotory and ran it in a container. In the current context we run the 32 bit Linux image and set it executing `bash`. At this point the prompts in the terminal window change so we can see we are running Linux in the container.

Containers are often describes as lightweight virtual machines, and this is the impression we have at this point. We are working on a 32 bit Ubuntu in a terminal window, but on a 64 bit host.

The Frankwolf image did not include support for gcc[1]. To provide this we use the command line to install the additional packages required for compilation of our code, e.g. `gcc5, g++5, binutils` and `make`.

When this is complete we exit from the container by entering the command `exit` and post the updated image back to the repository as

`billstoddart/ubuntu32bit_gcc5:vsn0`

2.2 The Forth "home" directory.

Our `RVM_FORTH` uses the concept of a "home" directory. During meta compilation Forth source code and C packages invoked from the Forth nucleus are located relative to this home directory. Later, when Forth runs, additional packages containing definitions that are not included in the nucleus are also located relative to the home directory.

The way in which a newly minted Forth gets to know where its home directory is necessarily different when we build it on a container, and we will come to these details in due course.

2.3 Compilation in the container

We use a bash script containing a Docker `run` command to pull down our saved container image from the Docker repository and mount the Forth home directory (which in our case is located at `/home/bill/rvm/rvm`) at `/root/rvm` in the filesystem of the 32 bit Linux provided by the container. However, note that due to the expansion of a symbolic link the former appears as `/home/bill/Dropbox/bill/rvm/rvm/` in some scripts.

When we run a container we are able to give it a command to execute. In our case we set the container to run the bash script that compiles our Forth system. When this script terminates control returns to the Bash script running on the host.

[1] It has since been updated.

2.4 Installation of the Forth system

The binary produced by the above process is left in the home directory. It is executable as a Forth system, but is not yet findable in the search path for executables, and does not know where its home directory is. This is fixed by creating a script that invokes Forth and tells it where its home directroy is. This script is then moved into a suitable directory in the search path. Again, details will follow.

3 A closer look

3.1 Running our container, the details

To perform a meta comilation of the system we navigate to the Forth home directory on the host machine and invoke the script `./dmcomp` ("docker meta compile"). The script requires root privlages to run docker, and since the Forth home directory is not in the search path the invocation is via the command:

```
sudo ./dmcomp
```

The `dmcomp` script ensures that the Docker demon is running and removes any present containers which might otherwise interfere with running the current container. It then sets our container running, pulling the image from the docker archive if it is not present locally. with no-network connection and directs it to run `bash` on the script `/root/rvm/dcomp0` in the container.

Here is the contents of the `dmcomp` script.

```
#! /bin/bash
echo "Checking if docker demon is active"
FRED=$(systemctl is-active docker)
echo "response is"
echo $FRED
if
  [ "$FRED" != "active" ]
then
   echo "issuing command: systemctl start docker"
   systemctl start docker
fi
echo "Removing any present containers"
docker container prune -f
echo "Attempting to run bash in container: billstoddart/ubuntu32bit_gcc5:vsn0"
docker run -it --network none \
--name gcc5 \
-v /home/bill/rvm/rvm:/root/rvm \
billstoddart/ubuntu32bit_gcc5:vsn0 \
bash -c /root/rvm/dmcomp0
echo "Now we are back on the host in directory $(pwd)"
```

Within the `run` command we invoke a number of switches which have the following effects:

- `-it` Specifies that the container should have an interactve terminal. This allows us to report progress and interact with the container if an error condition arises.
- `--network none` Specifies that the container should have no network connections.
- `-v ...` Specifies that the Forth home directory should be mounted at `/root/rvm` in the container file system.

Finally the line `bash -c /root/rvm/dmcomp0` Specifies the container should execute `bash` on the script `/root/rvm/dmcomp0`. This is the script `dcomp0` in the Forth home directory, now found at `/root/rvm/dmcomp0` in the container's file system.

3.2 Running 32 bit compilation within the container.

The script `dmcomp0` invoked as described above contains the instructions for metacompiling the Forth system and compiling associated gcc libraries. The only point to note is that it must navigate to the Forth home directory on the container Linux, since it has not been invoked from that directory. So the script `dmcomp0` begins as follows:

```
cd /root/rvm #Navigate to Forth home directory
#A script to build RVM_FORTH
 ...
```

When the script terminates so does the Docker `run` command invoked in `dmcomp` and control returns to that script which is running on the host machine..

3.3 Distribution and installation of RVM_FORTH.

When Forth is built on a container it does not get to know where its home directory is.

In our case the binary executable `4TH_BIN` is given this information by being run in the script `RVM_FORTH` with the command:

```
4TH_BIN SET-HOME /home/bill/Dropbox/bill/rvm/rvm/ $@
```

When `4TH_BIN` executes it runs the Forth interpreter on the following text, which provides commands to set the `RVM_HOME` directory.

For a different user, the home directory will be in a different place. We provide an `install` script which is run just once when `RVM_FORTH` is first installed. Typical usage is:

```
sudo ./install /usr/bin
```

which would create the appropriate `RVM_FORTH` script and move it to `/usr/bin`

The contents of the `install` script are as follows:

```
#! /bin/bash
#Run this script when RVM Forth is first installed.
#It writes the script RVM_FORTH which invokes Forth via the binary
#4TH_BIN and sets its home directory. It places the script in the
#directory given by $1
#A typical invocation could be:   install /usr/bin
#It does not need to be re-run each time Forth is meta-compiled.
#echo "STARTUP=$(pwd)/4TH_BIN SET-HOME $(pwd)/ \$@" "
STARTUP="$(pwd)/4TH_BIN SET-HOME $(pwd)/ \$@"
echo "STARTUP = $STARTUP"
echo "$STARTUP > RVM_FORTH"
echo $STARTUP > RVM_FORTH
echo "Move the newly created script to the given directory"
echo "mv RVM_FORTH " $1
mv RVM_FORTH $1
echo "set execute permissions"
echo "chmod a+x" $1"/RVM_FORTH"
chmod a+x $1/RVM_FORTH
```

4 Problems

We were initially working on a Ubuntu host, but the version of Docker installed by the Ubuntu package manager had a bug which made it impossible to mount a directory from the host within the container Linux file system. The workaround was to provide the Forth system to the container Linux in the form of a tarball. We found this too clumsy to be our preferred solution. Attempting to uninstall Docker and install the version provided on the Docker website broke the package manager. We resolved the problem by switching to Docker on Fedorer.

5 Conclusions

Containers are an interesting technology that can ease problems of software instability on continually evolving systems. In our application they enabled us to use a 32 bit gcc compiler which is no longer supported in current versions of Linux.

Acknowledgement

Warm thanks to Gerald Wodni for taking an interest in our system maintainence problems and pointing us in the direction of containers.

EuroForth 2024

A Forth binding for GTK4

Abstract

For some years, a Forth binding to major version 3 of the widely used graphical user interface toolkit GTK has been available. The major version 4 of GTK introduces many incompatibilities, so that a completely different approach to the binding is needed. It will be shown how the unique features of Forth can be leveraged to overcome the difficulties introduced by GTK4.

N.J. Nelson B.Sc. C.Eng. M.I.E.T.
Micross Automation Systems
Unit 6, Ashburton Industrial Estate
Ross-on-Wye, Herefordshire
HR9 7BW UK
Tel. +44 1989 768080
Email njn@micross.co.uk

1. Introduction

GTK is a popular toolkit for user interfaces. It was introduced in 1998 and remains under active development.

The developers have a policy of not maintaining backward compatibility across major versions, in favour of innovation.

Bindings and wrappers for major versions 2 and 3 have been available for some time, for VFX Forth.

The current version 4 of GTK has some very useful new features, which encourage application developers to switch to this version.

Unfortunately, in creating version 4, the developers of GTK made no provision for a relatively unusual language such as Forth. This resulted in the need for major changes in the bindings and wrapper code. Some particular difficulties have been solved in a way that would be possible only in Forth.

2. Overview of existing technique - GTK3

Application GUI elements such as windows and dialog boxes are designed using an interface builder program. Although there have been several of these available over the years, in practice only the program "Glade" was developed to full functionality. this produces XML type files with the .glade extension.

At an early stage in the compilation process, after compiling the GTK bindings and wrappers, the glade files are read in using functions in the GtkBuilder class. This creates all the graphical elements.

A Forth wrapper word then scans the list of all these elements and creates a Forth VALUE word for every named element. These words can then be used later in the code to manipulate the GUI, and to define all the callbacks using the CallProc: function.

Just as with Windows, GTK interacts entirely using callbacks. When all the GTK code has been compiled, the signals (e.g. button pressed) that were specified in the Glade design, can be automatically connected to the corresponding Forth CallProc: function, using another GtkBuilder function.

The main application window is then displayed. The function gtk_main word is then called, to run an infinite loop, passing signals and events as required, until one of them called gtk_main_quit. This was in accordance with the example code originally given by GTK3.

As GTK can operate only in a single thread, in order to preserve Forth interactivity while GTK is running, an extra thread was created to accept Forth input from the terminal.

3. Changes to the technique required for GTK4

3.1 Changes to the Interface Builder Program

The Glade interface builder program cannot produce XML files that are compatible with GTK4. Instead, a new builder program "Cambalache" (by the same developers as Glade, but looking rather different) can be used. This uses an intermediate file format - the format with a .ui extension compatible with GTK4 is produced using export. An interesting feature is that the builder components are more modular within Cambalache itself - the user interfaces for even a complex application could be accommodated into a single XML file. On the other hand, external modularity would be lost.

3.2 Changes to GtkBuilder

There is a major and most unfortunate change to the GtkBuilder class. It attempts to connect signals at the same time as creating the elements. This is non optional. By default, it tries to connect using the C symbol table. To accommodate other languages, there is a new class "GtkBuilderScope", but this is only briefly documented, with no examples. Other languages have already implemented GTK4 support, so I chose the open source language Rust to investigate how they implemented signal bindings. Unfortunately, the code for GtkBuilderRustScope is very complex. It did not look like the kind of solution that a Forth programmer would devise.

3.3 Changes to the GTK main loop

As GTK3 developed, the example code changed to recommend the use of a new GtkApplication class, which hid the main loop functions. The stated aim was the make it easier for application developers, but of course for a language like Forth this technique could not be used because Forth itself is always the application. It was most disconcerting to discover that in GTK4, the main loop functions gtk_main and gtk_main_quit had actually disappeared.

3.4 Changes to widget naming

In GTK3, widgets could optionally have an ID and / or a "name". The ID was used to refer to the widget in code. The second was used to uniquely identify the widget in CSS. Confusingly, the function gtk_buildable_get_name actually returned the ID.This function has disappeared in GTK4. This was very concerning because if we were unable to get at the name, we could not automatically create Forth VALUEs.

4. New solutions

4.1 Forth style solution to the signal connection problem

The issue about GtkBuilder performing a mandatory attempt at signal connection at the same time as creating the GTK widgets seemed to be insoluble. After a great deal of thought, we analysed our existing workflow, as follows:

a) Define the signal in Glade, usually naming it using the convention on_<widget-id>_<signal name>.
e.g. on_mybutton_clicked

b) Define the signal action, for the example above, typically:
2 0 CallProc: on_mybutton_clicked { pbutton puser -- }
...
;

We realised that we were typing the on_mybutton_clicked twice, and that the parameter list is duplicated in the 2 0 and the { pbutton puser -- }. Duplication of effort is not a Forth thing to do.

Perhaps, we could turn the problem to our advantage, by eliminating the signals completely from the Cambalache design program and instead, inventing a new word for signal definitions, using techniques that could only be done in Forth. As always, we looked at the desired end result first.

```
MYBUTTON CLICKED SIG: { pbutton puser -- } \ Defined the action for mybutton clicked
...
;

...

: MAIN ( --- ) \ Main function
...
  CONNECTUISIGS      \ Connect UI signals
...
;
```

Notice that in the example above, we save around 20 characters of typing per signal. This does not sound much, until one realises that in our main application, there are 594 signal definitions, and if we had used this technique from the start, we could have saved around 12,000 characters of code.

In order to implement this, we first have to define each type of signal, in the Gtkbindings.fth file.

```
\ ******************************************************************
\ Signal descriptions - all are ( ---z$,numins,numouts )
\ ******************************************************************

: CLICKED       Z" clicked"     2 0 ;       \ GtkButton clicked

...
```

Now we can define the signal handler word.

```
VARIABLE SIGLIST        \ Root of UI signal list

: SIG: { pwidget psigname pins pouts | pelement pentry pcb pxt -- }
\ Compiles signal response code and adds to connection list
  pwidget GETUIELEMENT -> pelement                      \ Get UI element from value
  pelement 0= ABORT" Unrecognised widget in SIG:"       \ Only valid UI elements
  HERE 7 CELLS+ -> pentry
\ 1 for link, 3 for chain data, 3 for callback prelude = 7
  SIGLIST LINK,                                         \ Add address to signal list
  pelement ,                                            \ Compile UI element
  psigname ,                                            \ Compile signal name
  pentry   ,                                            \ Compile callback entry
\ Make callback data structure, save structure address
  pins pouts CALLBACK, -> pcb
  :NONAME -> pxt                                        \ Compile the sig definition
\ Insert the call to the signal definition into the callback structure
  pxt pcb @ SET-CALLBACK
;
```

There is actually one redundant element in the chain data, in that the entry address is numerically related to the link itself. We have retained this for readability.

A slight complication is that in our application, we need to do things when we detect user inactivity (for example, automatically log users off). This requires a hook in the callback prelude in VFX, which results in a change in the magic number "7" in the above code. We will be allowing for this by making a deferred word for the entry position, so that it can be easily redefined when the hook is inserted.

Now we define the signal connection word, using the signal list we have created.

```
: CONNECTUISIGS ( --- ) \ Connect UI signals
  SIGLIST @                     \ Start of signal list
  BEGIN
  ?DUP WHILE                    \ There are still items in the list
    DUP CELL+                   \ Pointer to signal data
    DUP @ EXECUTE               \ Get object
    OVER CELL+ @                \ Get signal name
    ROT 2 CELLS+ @              \ Get callback entry point
    0                           \ User data (none)
    g_signal_connect DROP       \ Connect signal
    @                           \ Move to next item in list
  REPEAT
;
```

Note that the above solution does not allow for user data in the signal definition. All GTK signals allow for this, though it is required in only about 5% of actual code. A slightly extended SIGUSER: word could be defined, with an additional element in the chain for the user data, and SIG: could then be redefined as 0 SIGUSER:.

4.2 Maintaining interactive Forth during GTK execution

Instead of "going by the book", we looked more deeply into what was needed for both an interactive (debug) program, and an executable.

For the interactive version, we went back to the same technique that was used for the very first binding, for GTK2, which was written by MPE. Despite the known aversion to hooks in VFX, one is actually provided, deep in the interpreter, specifically for pumping user interfaces. For example:

```
: key-xterm \ sid -- key
  { | temp[ cell ] -- }
  begin
    dup key?-xterm 0=
   while
    emptyidle 2 ms
  repeat
  temp[ 1 rot read-xterm drop
  temp[ c@
;
```

Where EMPTYIDLE is a deferred word, defaulting to NOOP.

Now we can just hook up a message pump.

```
: GTKEMPTYIDLE ( --- )
\ While messages and GTK events are available, process them - used in key etc.
  BEGIN NULL FALSE g_main_context_iteration 0= UNTIL
;

: INSTALLGTKHOOK ( --- )
\ Install the GTK4 version of the message pump in the Forth interpreter
  ASSIGN GTKEMPTYIDLE TO-DO EMPTYIDLE
;
```

For the executable version, we can simply recreate a much simplified version of gtk_main.

```
: GTKMAIN ( --- ) \ Run the GTK main loop outside the Forth interpreter
  BEGIN
    gtk_window_get_toplevels g_list_model_get_n_items 0>
  WHILE                                          \ There are any top level windows
    NULL TRUE g_main_context_iteration DROP      \ Process messages and events
  REPEAT
;
```

Now select which one to use, in our MAIN word.

```
: MAIN ( --- ) \ Main function
...
  DEBUGGING IF              \ In debug mode
    INSTALLGTKHOOK          \ Install the message pump in the Forth interpreter
  ELSE
    GTKMAIN                 \ Run the GTK main loop outside the Forth interpreter
  THEN
;
```

4.3 The widget naming problem

Fortunately, there is now a function gtk_buildable_get_id, which now does what it says it does. Therefore, the automatic creation of Forth VALUEs is almost unchanged from GTK3. This was described in a previous paper but it's worth repeating because it is a very useful technique.

```
: MAKEUINAMES { | pslist pobject pname -- } \ Create values for every builder object
  PBUILDER gtk_builder_get_objects -> pslist          \ Make list of objects
  pslist g_slist_length 0 ?DO                         \ For all objects
    pslist I g_slist_nth_data -> pobject              \ Get data
    pobject gtk_buildable_get_buildable_id -> pname   \ Get name
    pname Z" ___" 3 S= 0= IF                          \ Filter out unnamed objects
      pname pobject ZVALUE                            \ Create value for each name
      pobject pname gtk_widget_set_name               \ Set widget name to be same
    THEN
  LOOP
  pslist g_slist_free                                 \ Free list
;
```

One other change from GTK3 is that all widgets now have IDs - if no ID is defined in Cambalache, the builder creates one automatically, and returns this in the object list. The automatically generated names all start with three underscores, and since we have no use for these in Forth, we can discard them.

5. Other changes required

We have described above the issues that are particular to Forth. Upgrading an application from GTK3 to GTK4 is still a major undertaking. The documentation lists 20 things that can be done in GTK3 to prepare for the change, and 93 things that must be done at the point of change. Of course, for any particular application not all of these will be relevant.

In the VFX GTK4 bindings file, we have commented out using
\ ** extern: ...
all the functions that have been deleted from GTK4 and where we will have to rewrite code to use new functions. There are 53 of these functions in our code. Some of them are in our code only once or twice, but some are called on a very large number of occasions.

We have allocated three man-months to the conversion.

6. Special Forth techniques that still work in GTK4

6.1 Internationalisation

In past papers I have described the techniques we use for internationalisation, with on-the-fly translation of the user interface. The base phrase (for example, of a label) is defined in Cambalache.

When the application starts, we can go through the following process:

a) Get the object list provided by GtkBuilder
b) Identify the type of each object (e.g. GtkLabel) as potentially requiring translation
c) Ask the object in its type appropriate way for its base phrase
d) Check that it is marked as a phrase that needs to be translated
e) Check whether that phrase is already in our database
f) If not, inserrt it into the database
g) Add the phrase reference number to the object
h) Set the phrase in the currently selected language into the object in the type appropriate way.

In a similar way, when the user requests a change of language, we just read back the phrase reference number from the object, and set the phrase for the new language.

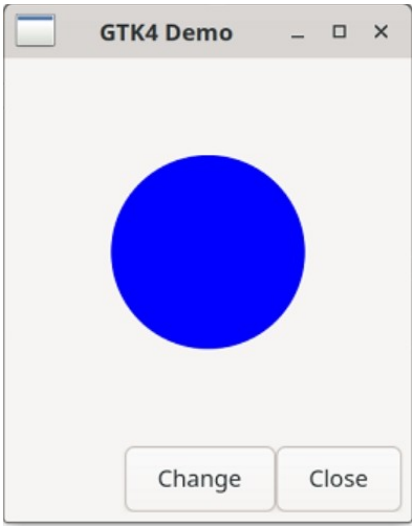
The good news is, all of these techniques appear to still work correctly in GTK4, even though one of the necessary functions is not directly documented - it is in the header file as a macro.

6.2 Drawing using Cairo

The appears to have been no change in the ability to use Cairo Graphics within GTK4, even though the widgets themselves appear now to be rendered using OpenGL, with Vulkan in progress.

7. The demonstration program

As with GTK3, we prepared a very small demonstration program to illustrate the basic principles.



The build and application files:

```
\ 64 bit floating point is required for Cairo

' REC-NDPFLOAT FORTH-RECOGNIZER \ Remove default floating point recogniser
-STACK
REMOVE-FP-PACK                  \ Remove default floating point package
include FPSSE64S.fth            \ Install SSE64 floating point
$26 -> ignSSEmask               \ Mask out floating point division by zero flag

TRUE CONSTANT DEBUGGING         \ Change when compiling an executable

include gtk4.fth                \ GTK4 tools
include DemoUI.fth              \ List of UI files for this application
include Cairo.fth               \ Cairo graphics file
gtk_init                        \ Initialise GTK
LOADUIS                         \ Load as part of compilation process
MAKEUINAMES                     \ Make values for each object
include gtk4demo.fth            \ Main program

.BadExterns cr                  \ Report any library failures
checkdict                       \ Report any dictionary corruption

GTK4DEMO                        \ Start demo program
```

```
1 1 CallProc: DEMOTIMER { pval  -- f } \ 2s timer event handling
  CURRCOL CASE                          \ Next colour
    0 OF 1 ENDOF
    1 OF 0 ENDOF
    2 OF 3 ENDOF
         2 SWAP
  ENDCASE -> CURRCOL
  demodrawingarea gtk_widget_queue_draw   \ Redraw drawing window
  TRUE                                    \ Return processed flag
;

5 0 CallProc: DEMODRAW { pwidget pcr pwidth pheight puser -- } \ Draw function
  pwidget -> CWIDGET          \ Set current widget
  pcr -> CCR                  \ Set current cairo context
  CURRCOL CASE                \ Select fill colour
    0 OF RED    ENDOF
    1 OF BLUE   ENDOF
    2 OF GREEN  ENDOF
         ORANGE SWAP
  ENDCASE BRUSH
  50 50 150 150 ELLIPSE       \ Draw a circle in the middle
;

DEMOCLOSEBUTTON CLICKED SIG: { pbutton puser -- } \ Close button clicked
  demowindow gtk_window_destroy
;

DEMOCHANGEBUTTON CLICKED SIG: { pbutton puser -- } \ Change button clicked
  CURRCOL 0 1 WITHIN IF 2 ELSE 0 THEN -> CURRCOL \ Switch colour pair
;

: GTK4DEMO-WINDOW ( --- ) \ Initialise and show demo window
  demodrawingarea DEMODRAW NULL NULL gtk_drawing_area_set_draw_func \ Set function
  demowindow gtk_window_present                                     \ Show window
  2000 DEMOTIMER 0 g_timeout_add DROP                               \ Start timer
;

: GTK4DEMO ( --- ) \ Main function
  #BADLIBS IF                       \ Any libraries not loaded
    Z" Libraries not loaded" FATAL
  THEN
  PBUILDER 0= IF                    \ UI not loaded ( in executable mode )
    gtk_init                        \ Initialise GTK
    LOADUIS                         \ Load UI files
    SETUIVALS                       \ Set values for UI objects
  THEN
  CONNECTUISIGS                     \ Connect UI signals
  gtk_init                          \ Initialise GTK
  GTK4DEMO-WINDOW                   \ Initialise and show demo window
  DEBUGGING IF                      \ In debug mode
    INSTALLGTKHOOK                  \ Install the pump in the Forth interpreter
  ELSE
    GTKMAIN                         \ Run the GTK main loop outside interpreter
  THEN
;
```

8. Conclusion

In Forth, there is always a way to overcome the problems that library developers inadvertently create.

Creating a new Forth application in GTK4 is very straightforward, however, updating an existing application from GTK3 to GTK4 is a major undertaking, whichever language you use.

Pac-Man for the DEC VT420

François Laagel *
Institute of Electrical and Electronics Engineers

Abstract

Pac-Man is a graphical game designed in 1979 by a team of five people and implemented in Z80 assembly language over the course of seventeen months. This article is an evolutionary account of my own Forth implementation in ANS94 Forth for the Digital VT420 text terminal over a three month period. The C port of the resulting application to Linux and OpenVMS 9.2 will also be briefly covered.

Stress will be laid upon the value of standards throughout this paper. Various development/prototyping tools were required for this implementation to be successful. In essence, this paper is a first person account of an experience in retrocomputing.

1 Background and Motivation

Z79Forth is a single board computer I started designing by the end of 2018. It is based on the Hitachi HD6309 microprocessor, a much improved implementation of the Motorola MC6809. It runs three to five times faster, uses less current and has an extended instruction set.

The firmware I developed for the board is a Forth operating system originally written for the Zilog Z80 back in 1983. Two Git branches are available, allowing an end user to select either a 79-STANDARD subset or an ANS94 Core implementation. The latter was used as a target for this artful endeavour (see also [1]). GNU Forth 0.7.3 under Linux was used as a development platform. To that end, a very high degree of compatibility was essential.

During firmware development, flow control over the serial asynchronous communication line via USB (FTDI-232RL based) was found to be problematic at times. UART management was initially strictly *programmed IO* based. Yet even after switching to an interrupt based scheme, characters were occasionally lost during "cut & paste" of large text chunks. So, I ended up going for an alternate connectivity option over RS232. In 2020, I acquired a DEC VT420 terminal so as to be able to communicate with the board without having to resort to a dedicated frontend system. Worth noticing is the fact that DEC terminals do not support hardware flow control (RTS/CTS based) but implement software flow control (XON/XOFF) instead.

The DEC VT420 [2] is a technological wonder of the nineties. It is a monochrome Intel 8031 based text terminal equipped with an 800 by 400 pixels display. Most importantly, it supports user defined fonts. Which is the reason why I came across the idea of implementing Pac-Man specifically for it.

2 Research

Pac-Man is the brainchild of Toru Iwatani. Iwatani [3] was self-taught in computers without any formal training in programming or graphic design. The game aimed to appeal primarily to women and was originally released in Japan in July 1980.

Pac-Man is driven by the player (via a joystick) and is free to move within the confines of a maze. Every reachable spot of the maze has to be visited at least once. Four ghosts, initially located inside of a pen at the center of the maze, compete against the player for his/her life. Throughout the maze *collectible* items are interspersed. These are either *dots* or *power pellets*, which carry a greater point value and grant Pac-Man some temporary immunity against the ghosts.

Every substantial project begins with reference material collection. Fortunately, Pac-Man has been reverse engineered down to the level of implementation bugs analysis. The ghosts moving strategy was an early concern of mine. The **Gameinternals** web site [4] supplies a thorough documentation with respect to this. The ultimate reference document remains Jamey Pittman's "**Pac-Man dossier**" [5].

In 1995, Roar Thornaes [6] released a C++ based Pac-Man implementation targeting X11/POSIX

*f.laagel@ieee.org

under Linux or CYGWIN under Microsoft Windows. I selected his maze's topology–which differs from that of the original game and decided that Thornaes' object oriented approach was the way to go. To that end, I elected to go for GNU Forth 0.7.3's API to object orientation–which was later on made an integral part of the Forth2012 standard [7]. I ended up adopting the specification literally, using a cross-platform development model from GNU Forth 0.7.3 and targeting, ultimately, **Z79Forth/A**. The convenience of adherence to standards for this purpose cannot be emphazised strongly enough.

3 Methodology

The development approach I followed was originally a *shot in the dark*. In retrospect, it still looks to me as a very rational way to conduct such an ambitious project.

Presentation Layer This entailed figuring out how to draw a workable playing screen on a text based terminal. An essential part of this was to come up with a nice looking user defined font, so as to be reasonably faithful to the original implementation.

Basic Business Layer This covers object identification and definition. It also includes the implementation of a very basic gameplay. Initially:

- A shared ghost moving/display policy–mostly based on random direction selection. That policy also addressed the delicate subject of preserving erasable characters items on screen.
- A specific Pac-Man moving/display policy, which had at some point to address gathering collectibles and its gobbling/non-gobbling state. It also handles score management, collision detection and remaining lives count maintenance.

Business Layer Successive Refinements At this point, differentiated moving strategies for each of the four ghost instances were implemented. This also implied *ghost mode* management. At any given time, the ghosts can be in any of the following three states: *scatter*, *chase* or *frightened*. A finite, partially time based, state automaton drives the changes between those states.

Testing This is the fun part, of course. Bug fixing also comes with the territory!

Specific Support for the VT340 The VT340 has a different matrix size for user defined font specification. The gameplay code is entirely shared with the VT420.

Publish the Forth Code as public domain software on Github.

Linux/C Port I am not aware of any working Forth implementation for OpenVMS. So I went for a straight port to C.

OpenVMS/C Adaptations OpenVMS is mostly POSIX compliant but there are some differences.

To do: Validate the C code on OpenVMS and publish it.

4 VT420 Font Development

At some point Pablo Hugo Reda sympathized with the utterly futile nature of my project and suggested *The spriters resource* [8] as a valuable starting point.

When I consulted him for guidance about sprite design, Pablo mentioned *Piskel* [9] as the tool he uses in the context of his side teaching activity. He also said he designed his own sprites manually. Obviously, I took his input and set out to develop my own custom font for the VT420. This turned out to be a tedious and very time consuming operation, which ended up taking me some twelve full-time working days.

The original Z80 based implementation resorted to a 224 by 288 graphical display [10]. Because I targetted a monochrome text terminal, I first had to let the colors go and, instead, focus on distinctive looking aspects for each of the four ghosts. Pac-Man himself is supplied with a visible eye which was not among the features of the original game.

Typical text display terminals of the 1990's offered an 80 x 25 display capability. The VT420 is a renowned member of that class of devices. Because user-defined fonts are specified, in sixel terms, as a bitmap matrix 10 pixels wide and 16 pixels high, I chose to operate in a similar mode, which amounts to 33 columns double width characters by 23 physical rows.

This resulted in a font of 30 double width characters. Figure 1 illustrates the bitmap design

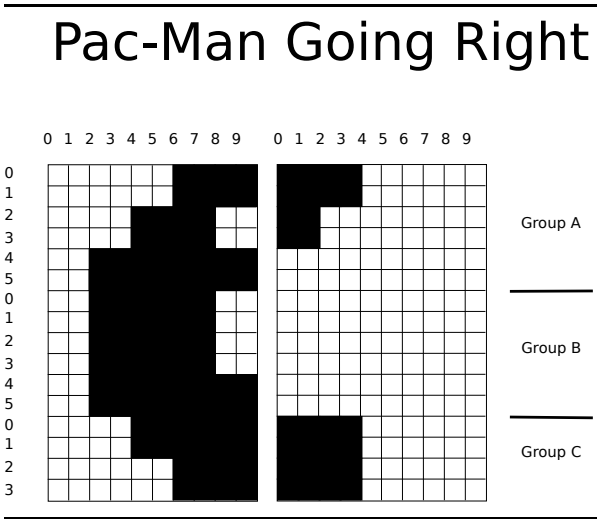


Figure 1: Pac-Man Going Right Sprite

```
CREATE softfont
[...]
\ Character "Pacman going right", left half at $25 (%).
\ Left: column #0 and #1 unused.
\ Group A  Group B    Group C (top to bottom)
%000000 C, %000000 C, %0000 C, \ Column #0 unused
%000000 C, %000000 C, %0000 C, \ Column #1 unused
%110000 C, %111111 C, %0000 C, \ Column #2 (eff. 0)
%110000 C, %111111 C, %0000 C, \ Column #3 (eff. 1)
%111100 C, %111111 C, %0011 C, \ Column #4 (eff. 2)
%111100 C, %111111 C, %0011 C, \ Column #5 (eff. 3)
%111111 C, %111111 C, %1111 C, \ Column #6 (eff. 4)
%111111 C, %111111 C, %1111 C, \ Column #7 (eff. 5)
%110011 C, %110000 C, %1111 C, \ Column #8 (eff. 6)
%110011 C, %110000 C, %1111 C, \ Column #8 (eff. 7)

\ Character "Pacman going right", right half at $26 (&)
\ Right: columns #8 and #9 unused.
\ Group A  Group B    Group C (top to bottom)
%001111 C, %000000 C, %1111 C, \ Column #0 (eff. 8)
%001111 C, %000000 C, %1111 C, \ Column #1 (eff. 9)
%000011 C, %000000 C, %1111 C, \ Column #2 (eff. A)
%000011 C, %000000 C, %1111 C, \ Column #3 (eff. B)
%000000 C, %000000 C, %0000 C, \ Column #4 (eff. C)
%000000 C, %000000 C, %0000 C, \ Column #4 (eff. D)
%000000 C, %000000 C, %0000 C, \ Column #4 (eff. E)
%000000 C, %000000 C, %0000 C, \ Column #4 (eff. F)
%000000 C, %000000 C, %0000 C, \ Column #8 unused
%000000 C, %000000 C, %0000 C, \ Column #9 unused

\ ------------------------------------------------------------
\ DECDLD spec:
\ DCS Pfn; Pcn; Pe; Pcmw; Pss; Pt; Pcmh; Pcss { Dscs
\ Sxbp1 ; Sxbp2 ;...; Sxbpn ST

1  CONSTANT pfn   \ Font number
1  CONSTANT pcn   \ First soft char at $21--do not override BL!
1  CONSTANT pe    \ Erase only new definitions
10 CONSTANT pcmw  \ Character width is 10 pixels in 80 col mode
0  CONSTANT pss   \ 80 columns, 24 lines
2  CONSTANT pt    \ Full cell
16 CONSTANT pcmh  \ Character height is 16: 6/6/4 in sixels
0  CONSTANT pcss  \ 94-charset, $20 (BL) and $7E (~) are RO
85 CONSTANT ufn   \ User font name is 'U' (argument to DSCS)

\ See: https://vt100.net/docs/vt420-uu/chapter9.html
\ for full parameter description.
??oo{{~~rr/??~~~~~~oo/????BBNNNN; \ Char $25 (%) definition
NNBB??????/??????????/NNNN??????  \ Char %26 (&) definition
```

Figure 2: Pac-Man Going Right Encoding

for the use case *Pac-Man Going Right*. Figure 2 shows the corresponding Forth code, an outline of the VT420 font definition protocol (DECDLD) and what it amounts to in ASCII terms.

The concept of virtual rows originated from the notion that, somehow, the aspect ratio of the original game could be preserved, from a timing perspective. Conceptually, a virtual row is twice the physical line number that the terminal can directly address. The game's main engine works by assuming a predictable run time environment and updating the game's current state based on the notion that the game actually works in a grid space that is four times larger than it actually is (as visualized on the terminal): double-width characters combined with a divided by two scheduling on the verticals. Basically, the columns are handled physically and the rows virtually.

5 An Object Oriented Approach

Although my approach to object orientation is rather primitive–there is no need for inheritance support and an object is little more than a way to store state information in a centralized way–it has proven to be effective to solve the problem at hand. The central concept is that of an *entity*. An entity is an object subjected to various *methods*. There are five entity instances: Pac-Man is described in an entity vector which resides–by convention–at offset 0. The other entities are each of the four ghosts: **Blinky**, **Inky**, **Pinky** and **Clyde**. Pointers to these reside in the same vector. Each entity has an instance number attribute that is used to implement differentiated behaviour (figure 3).

The methods implemented are:

- `strategy`: the word stored in that field is the address of a Forth execution token that determines how the object moves from one clock cycle to the next one.
- `display`: a pointer to a Forth word that displays the object, taking into account its current state attributes. This is invoked on every clock cycle, after the `strategy` method has been called.

6 Scheduling

Any kind of game requires some form of real-time user interaction. This can be accomplished through either an event-triggered approach or a time-based one. Micheal J. Pont [11] strongly advocates the latter, which relies entirely on cooperative task switching and a timer based interrupt handler that is supposed to be able to deal with any unexpected condition, should the need for this ever arise. I

```
BEGIN-STRUCTURE entity
  FIELD:  e.strategy \ Strategy (moving) method
  FIELD:  e.display  \ Display method
  CFIELD: e.resurr   \ # Clock ticks till we're back (ghosts)
  CFIELD: e.reward   \ # points for killing a ghost / 100 (PM)
  CFIELD: e.vrow#    \ Virtual row number
  CFIELD: e.pcol#    \ Physical column number
  CFIELD: e.ivrow#   \ Interfering virtual row number (ghosts)
  CFIELD: e.ipcol#   \ Interfering pcol number (ghosts)
  CFIELD: e.igchr    \ Interfering grid character (ghosts)
  CFIELD: e.pcol0    \ Initial pcol number
  CFIELD: e.vrow0    \ Initial vrow number
  CFIELD: e.dir0     \ Initial direction
  CFIELD: e.hcvr#    \ Home corner vrow# (ghosts)
  CFIELD: e.hcpc#    \ Home corner pcol# (ghosts)
  CFIELD: e.cdir     \ Current direction
  CFIELD: e.pdir     \ Previous direction
  CFIELD: e.idir     \ Intended direction (PM)
  CFIELD: e.revflg   \ Reverse direction directive (ghosts)
  CFIELD: e.inited   \ TRUE if first display has been performed
  CFIELD: e.gobbling \ # Clock ticks till we're fed (PM)
  CFIELD: e.inum     \ Instance serial number
END-STRUCTURE

\ Method invokator.
: :: ( method-xt-addr -- ) @ EXECUTE ;
```

Figure 3: Entity Object Specification

```
: _main ( -- )
  BEGIN
    \ Check if remitems# is 0 here. If so start new level.
    remitems# @ 0= IF
      .initial-grid
      update-level
      level-entry-inits
    ELSE
      entvec @ DUP e.strategy :: \ Pacman's move
      entvec CELL+ #ghosts 0 ?DO
        DUP @ DUP e.strategy :: \ Move the current ghost
        CELL+
      LOOP DROP
      clkperiod MS
    THEN
  AGAIN ;

: main ( -- )
  initialize
  entvec @ TO pacman-addr
  0 remitems# !     \ Force level entry initializations
  PAGE .init-sitrep \ Initial scoreboard

  IFZ7 _main finalize
  IFGF ['] _main CATCH finalize
  IFGF ?DUP IF
  IFGF   0 23 AT-XY ." Caught exception " DUP . CR
  IFGF   ." Depth was " DEPTH . CR
  IFGF   THROW      \ We still want a stack dump!
  IFGF THEN
;
```

Figure 4: Early Engine Code

decided to go for a time-based approach with no interrupt handler support–a reliable time basis remains essential to the game's playability though.

An early incarnation of the game's main engine is shown in figure 4. At that point, involving the `strategy` method still implied an implicit reference to the `display` method.

Z79Forth/A is not a multi-tasking system. Which means the firmware does not have to compromise against any other running task. Properly programmed, it makes the platform ideally suited for a real time type of application. Precisely what was needed under the present circumstances. The Linux kernel in its 5.4.0-150 kernel incarnation (Mint 19.3) is not real time by default and this presents a challenge that cannot be easily overcome. The *chrt(1)* manual page is so poorly written as to be almost incomprehensible by the common man. The default Linux scheduler class will eventually adapt itself to the actual application behaviour and end up doing the right thing–after some time, during which the player may very well be killed by the ghosts!

7 Ghost Mode Management

A scheduler can hide another one. Quoth the **Pac-Man Dossier**:

> Ghosts alternate between *scatter* and *chase* modes during gameplay at predetermined intervals. These mode changes are easy to spot as the ghosts reverse direction when they occur. Scatter modes happen four times per level before the ghosts stay in chase mode indefinitely. [...] The scatter/chase timer gets reset whenever a life is lost or a level is completed. At the start of a level or after losing a life, ghosts emerge from the ghost pen already in the first of the four scatter modes.

In *scatter* mode, the ghosts navigate to their home corners. They are:

Pinky	top left
Blinky	top right
Clyde	bottom left
Inky	bottom right

The *gospel* goes so far as to specify the reverse engineered state transition time table (expressed in elapsed seconds).

Mode	Level 1	Levels 2-4	Levels 5+
Scatter	7	7	5
Chase	20	20	20
Scatter	7	7	5
Chase	20	20	20
Scatter	5	5	5
Chase	20	1033	1037
Scatter	5	1/60	1/60
Chase	$+\infty$	$+\infty$	$+\infty$

At this point, I felt the need to develop a ghost mode simulator completely independent from the game itself. Ultimately, I validated the concept and integrated it into the game's engine. I originally implemented the delays in strict millisecond terms; it later proved more useful to express those as multiples of the scheduler's timeslice unit (`clkperiod`).

8 AI Integration

There is, of course, no such thing as artificial *intelligence*. The only intelligence that can be perceived by an end user is that of the programmers' and the size of their factual database–this results, at best, in *unpredictable outcome computing*. And yet, the various ghosts moving strategies, as outlined in the **Pac-Man Dossier**, still refer to this as *AI*. The bottom line is that, when the *chase* mode is in effect, each ghost instance has a specific policy that is involved every time a direction change is possible. That policy set is fairly simple yet conducive to a great gameplay experience.

Blinky targets Pac-Man's current location.

Pinky targets a position that is four tiles ahead of Pac-Man's current moving direction.

Inky targets the end of a vector twice as long as the one originating from **Blinky** to Pac-Man's moving direction extrapolated by four half tiles.

Clyde does not know what it's doing. Its direction changes are as unpredictable as the LFSR based random number generator. The latter is, by design, completely deterministic.

From a programming perspective, this all comes down to the primitive listed as figure 5.

```
\ For every bit set in bitmap, we need to evaluate the
\ Euclidian distance between the potential next location and
\ the target tile. Finally we return the direction that
\ minimizes the distance.
\ Note: U> is ANS94 'Core Ext', so I'm using it.
: ghost.dirselect-nav2target ( self bitmap tvr tpc )
  ( -- new-dir )
  3 ROLL 3 ROLL          \ S: tvr\tpc\self\bitmap
  0 65535                \ S: tvr\tpc\self\bitmap\minoff\minval
  dir_blocked dir_up DO
    2 PICK I bitset? IF  \ Direction I is an option
      3 PICK e.pcol# C@
      \ S: self\bitmap\minoff\minval\pc-cur
      4 PICK e.vrow# C@
      \ S: self\bitmap\minoff\minval\pc-cur\vr-cur

      I case!
      dir_left  case? IF SWAP 1- SWAP THEN
      dir_right case? IF SWAP 1+ SWAP THEN
      dir_up    case? IF 1- THEN
      dir_down  case? IF 1+ THEN
      \ S: tvr\tpc\self\bitmap\minoff\minval\pc-next\vr-next

      \ Selected target is at [tvr, tpc].
      7 PICK - DUP *
      \ S: tvr\tpc\self\bitmap\minoff\minval\pc-next\dy^2
      SWAP 6 PICK - DUP *
      \ S: self\bitmap\minoff\minval\dy^2\dx^2
      +                  \ We compare the squared distance
    ELSE
      65535              \ uint16 max
    THEN
    \ S: tvr\tpc\self\bitmap\minoff\minval\minval-new

    2DUP U> IF
      NIP            \ S: tvr\tpc\self\bitmap\minoff\minval-new
      NIP I SWAP         \ S: tvr\tpc\self\bitmap\I\minval-new
    ELSE
      DROP
    THEN
  LOOP                   \ S: tvr\tpc\self\bitmap\minoff\minval

  DROP NIP NIP NIP NIP ;
```

Figure 5: Ghosts' Navigation Code

9 VT340 Port

This presented no real technical difficulty. The 340 has a higher resolution than the 420 and user defined fonts on the 340 are specified via a 10 by 20 bitmap matrix, as opposed to a 10 by 16 one for the 420. To that end I devised a C based font scaler utility which saved me a lot of time.

I also gave *fake text colour* rendition a try using ReGIS (`Remote Graphic Instruction Set`). It turned out that the required overhead on both the terminal processing abilities and the limited serial line throughput did not make such an option viable.

10 OpenVMS C Port

Back in April 2023, *VMS Software Inc.* announced the release of OpenVMS 9.2 for hobbyists (see [12]), targeting the x86-64 platform. This was, of course, music to my ears and I jumped on the bandwagon as soon as I was able to.

I started with a low ball target: a Linux/gcc 7.5 implementation. I resorted to a 32 bit cell representation for compatibility with VMS. Here again, standards come in handy. POSIX.1 (see [13]) naturally comes to mind when it comes to Unix interoperability. I did base my C port on that specification and it works reasonably well under GNU/Linux Mint 19.3 (5.4.0-150 kernel).

I experimented with POSIX based Forth essential primitives support via a dedicated prototyping tool. These central words were only a simple subset: `MS ?KEY AT-XY KEY CR`. The game logic inherited from the original Forth code was entirely preserved.

Upon startup, Forth expects the terminal to be in raw mode. In POSIX terms, this is expressed as an ad'hoc **tcgetattr/tcsetattr** system call sequence. Under OpenVMS, things are slightly different in that this has to have a *libcurses* equivalent incantation (**noecho/crmode**).

VMS also departs from the POSIX specification, although in a very marginal way, since the `select` system call is not implemented as a standard system interface but as a part of the network API.

A final note regarding C coding standards: one should definitely not rely on the *0b* prefix for spec-

ifying binary literals in C. This is a `gcc` extension and it is in no way standardized!

11 Conclusions

The *final* Forth code is about 2400 lines long. This amounts to about 60 blocks of CompactFlash storage. The user defined font requires some 600 lines for the VT420. The rest of the material is divided about evenly between comments and effective code. This software has been pushed into the public domain and published on Github [14]. The C port is of the same *order of magnitude*.

It works reasonably well under **Z79Forth**, partly because system resources do not have to be shared between competing processes in this environment. The Linux/C port can also, somehow, behave itself in a satisfactorily manner, after a few seconds. The OpenVMS/C port has proven to be a radical failure due to my inability to enable software flow control on the target system.

This being said, under **Z79Forth**, the lack of firmware supported exceptions can lead to disconcerting program termination at times. Currently, the traditional Unix SIGINT (emitted when Control-C is parsed from the controlling serial communication line) ends up being routed to the error handler routine. The latter issues an ASCII *Shift In* control character, causing the terminal to revert to the default character set. And yet, at the time of this writing, it still fails to restore the cursor status as being enabled (a VT200 control sequence). It all comes down to being able to emit `<CSI>25h` when you have only nine bytes of EEPROM available.

User-level exceptions (based on Mitch Bradley's `CATCH/THROW` model) have been successfully prototyped on the platform. The underlying code remains experimental and unpublished. Ideally, this should be an integral part of the firmware as supplied by default. At this point, *it is clear that more research is required!*

Key takeaways:

- *Standards Matter.*
- *Ad'hoc Tools* also are essential and will cut down your development time substantially.

References

[1] Bret Victor
Stop Drawing Dead Fish
`https://www.youtube.com/watch?v=ZfytHvgHybA`

[2] Digital
VT420 Text Terminal, commercial brochure
Digital Equipment Corporation, 1990.
`https://vtda.org/docs/computing/DEC/Terminals/EC-F0682_VT420TextTerminalBrochure_1990.pdf`

[3] *Programmers at Work, Toru Iwatani*
Microsoft Press, 1986. Penguin Books, 1989.
ISBN: 1-55615-211-6.
`https://programmersatwork.wordpress.com/toru-iwatani-1986-pacman-designer/`

[4] Game Internals
Understanding Pac-Man Ghost Behavior
`https://gameinternals.com/understanding-pac-man-ghost-behavior`

[5] Jamey Pittman
The Pac-Man Dossier. August 11, 2015
`https://pacman.holenet.info/`

[6] Roar Thornaes
Source code repository for the C++ Pac-Man Implementation
`https://sources.debian.org/src/pacman/10-6/`

[7] Forth2012 Standard Committee
Standard Specification. FACILITY EXT word set, BEGIN-STRUCTURE
`https://forth-standard.org/standard/facility/BEGIN-STRUCTURE`

[8] The Spriters Resource
The VG Resource, 2024.
`https://www.spriters-resource.com`

[9] Piskel
An interactive online sprite design application
`https://www.piskelapp.com`

[10] Jamey Pittman
NAMCO Physical Platform Specifications, 2015.
`https://pacman.holenet.info/#Specifications`

[11] Michael J. Pont
Patterns for Time-Triggered Embedded Systems
ACM Press/Addison-Wesley, 2001
ISBN: 0-201-33138-1

[12] VMS Software Inc.
OpenVMS 9.2-2 Public Release Announcement
`https://vmssoftware.com/about/v922/`

[13] The Open Group
IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)
`https://pubs.opengroup.org/onlinepubs/9699919799/`

[14] François Laagel
Source Code Repository for "Pac-Man for the DEC VT420"
Github, January 2024.
`https://github.com/forth2020/frenchie68/tree/main/pacman`

How to Implement Words (Efficiently)

M. Anton Ertl*
TU Wien

Abstract

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

1 Introduction

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

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

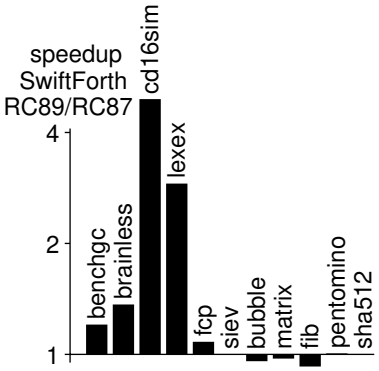


Figure 1: Speedup of SwiftForth 4.0.0-RC89 over SwiftForth 4.0.0-RC87 on a TigerLake CPU

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

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

2 Performance pitfalls

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

2.1 False sharing between I and D-cache

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

*anton@mips.complang.tuwien.ac.at

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

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

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

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

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

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

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

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

2.2 Return misprediction

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

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

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

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

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

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

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

3 Requirements

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

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

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

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

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

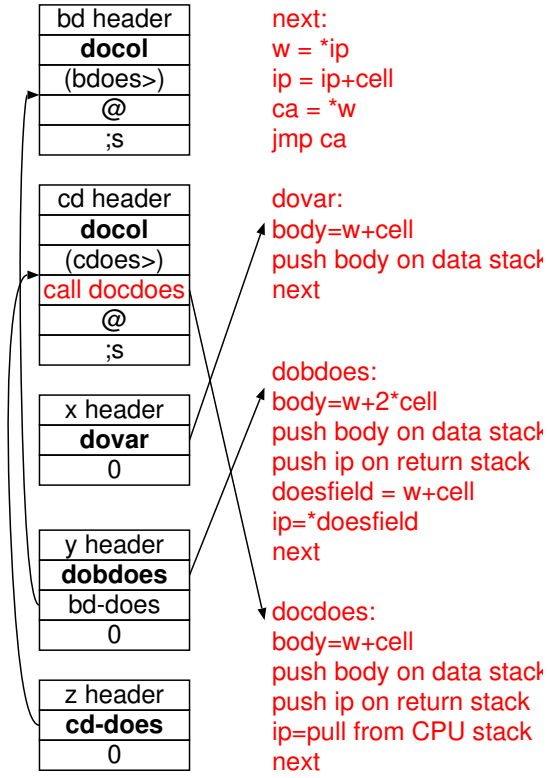


Figure 2: Implementation of words with associated data in indirect-threaded code. Code field in **bold**, native (pseudo-)code in red.

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

4 Indirect-threaded code

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

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

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

4.1 `Does>`

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

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

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

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

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

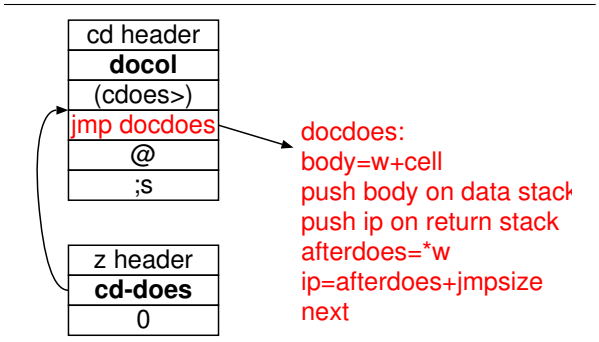


Figure 3: An implementation variant for `cdoes>` that uses a jmp instead of a call.

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

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

5 Alternative implementation techniques

5.1 Avoiding return mispredictions

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

5.2 Direct-threaded code (ITC style)

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

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

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

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

5.3 Subroutine-threaded code

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

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

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

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

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

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

5.4 Avoid body

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

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

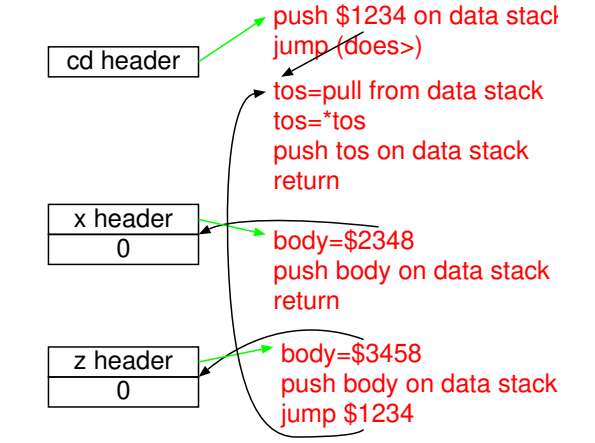


Figure 4: Trampolines for `x` and `z`. While the header points to the trampoline, this pointer is not followed at run-time (so it is not a code field), but at text-interpretation time. The code is shown as pushing and popping, but usually this works with registers

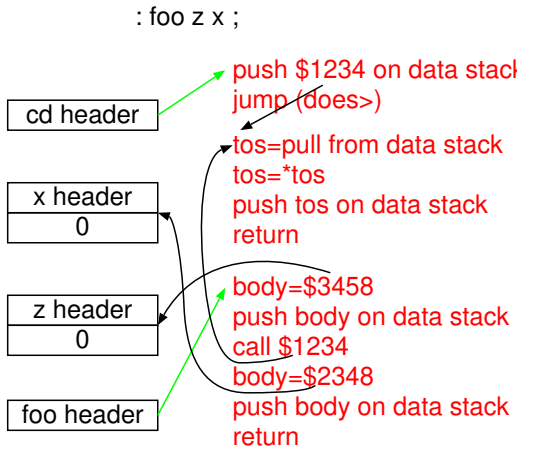


Figure 5: Code compiled for `foo` with an intelligent `compile,`.

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

```
: c 5 ;
```

5.5 Trampolines

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

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

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

5.6 Intelligent `compile,`

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

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

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

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

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

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

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

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

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

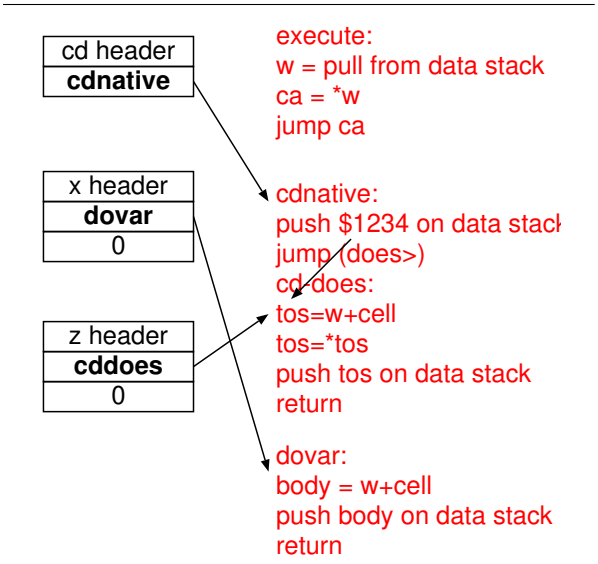


Figure 6: A native-code system with a code field containing a code address (as in ITC)

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

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

5.7 `Deferred` words

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

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

5.8 Native-code address field

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

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

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

5.9 Always have a doesfield

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

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

5.10 Double-indirect threaded code

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

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

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

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

6 Measurements

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

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

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

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

Here are the Forth words that the microbenchmarks measure:

```
create x 0 ,

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

d1 z1

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

d2 z2

0 constant my0

defer w ' my0 is w
```

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

6.1 The original problem

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

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

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

6.2 ... and it's `execute` variant

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

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

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

6.3 Is VFX always fine on compiled code?

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

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

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

6.4 What about iForth?

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

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

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

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

6.5 Compiled `created` words are fast

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

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

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

6.6 ... but once you `execute` ...

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

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

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

6.7 What about `defer` and `is`?

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

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

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

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

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

6.8 ... in combination with `execute`

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

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

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

6.9 What about `defer` without `is`?

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

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

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

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

6.10 ... in combination with `execute`

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

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

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

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

7 Related work

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

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

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

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

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

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

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

8 Conclusion

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

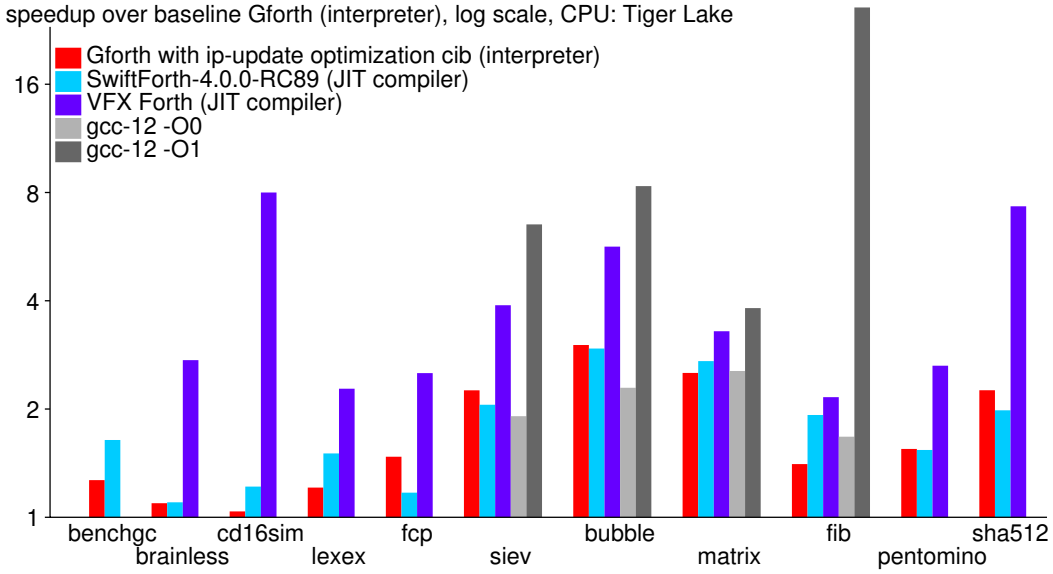
The Performance Effects of Virtual-Machine Instruction Pointer Updates 2024 update

M. Anton Ertl, TU Wien
Bernd Paysan, net2o

Overview

- Background: Virtual-machine interpreter with code copying
- Every VM instruction increments the VM instruction pointer (IP)
- Question: How relevant are IP updates for performance?
- Answer: on some programs critical latency path
- Method: optimize most IP updates away

Is interpreter performance relevant? What about JITs?



Running example: inner loop of `siev`

Forth Source:

```
do
  0 i c! dup +loop
```

C Source:

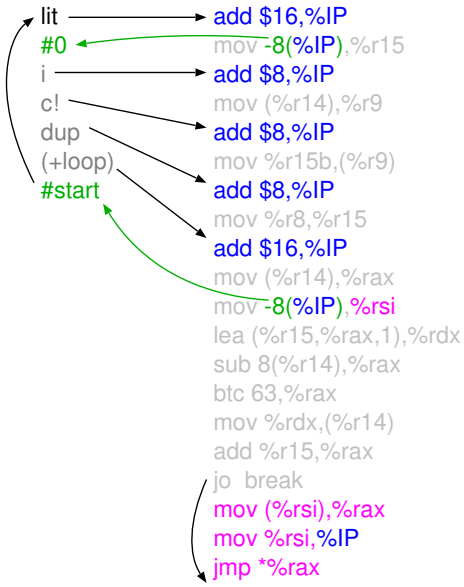
```
for(p = ... ; p <= ... ; p += prime)
  *p = 0;
```

Virtual-Machine code (Gforth):

```
        (do)
start:  lit
        #0
        i
        c!
        dup
        (+loop)
        #start
```

3

Baseline: Code-copying interpreter with static stack caching



It's a JIT compiler!

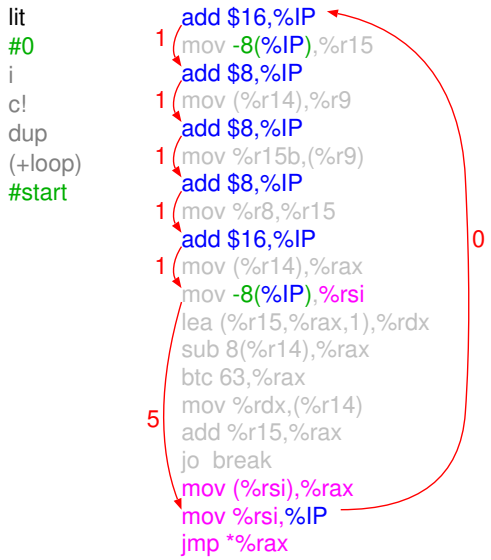
\+ Copies native code

It's an interpreter!

\+ Portable
gcc generates code snippets
\+ Fallback option
to threaded-code interpreter
without code copying
\+ VM code is still needed
for immediate values
for control flow
\+ ⇒ VM instruction pointer needed

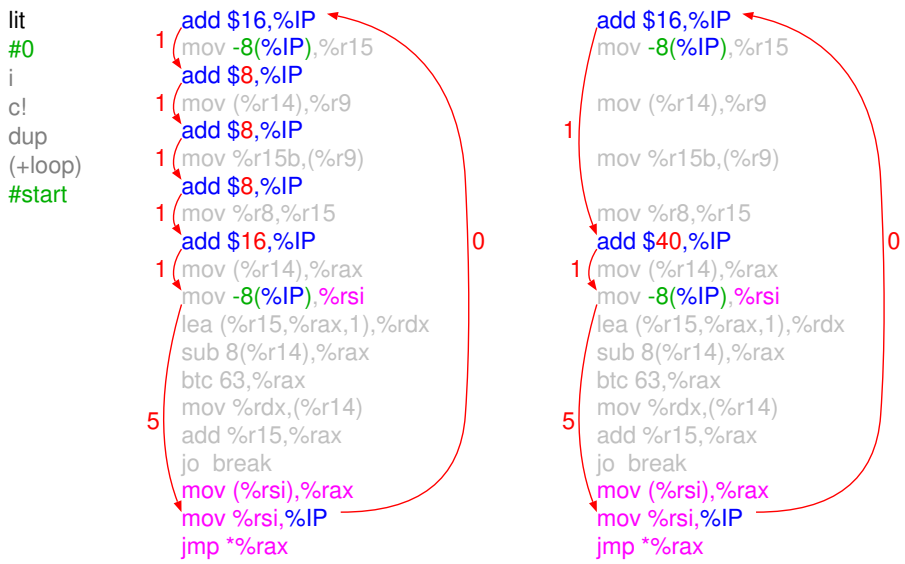
4

Instruction pointer updates limit execution rate

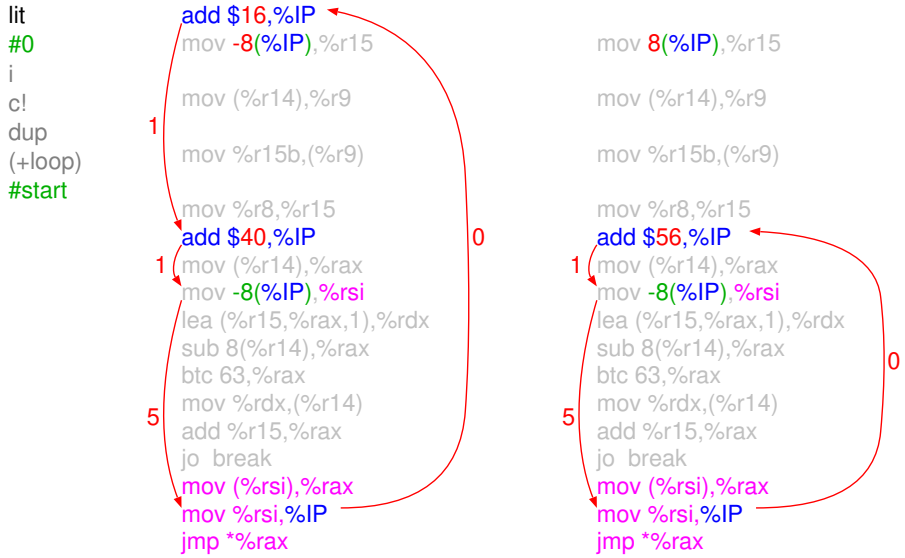


5

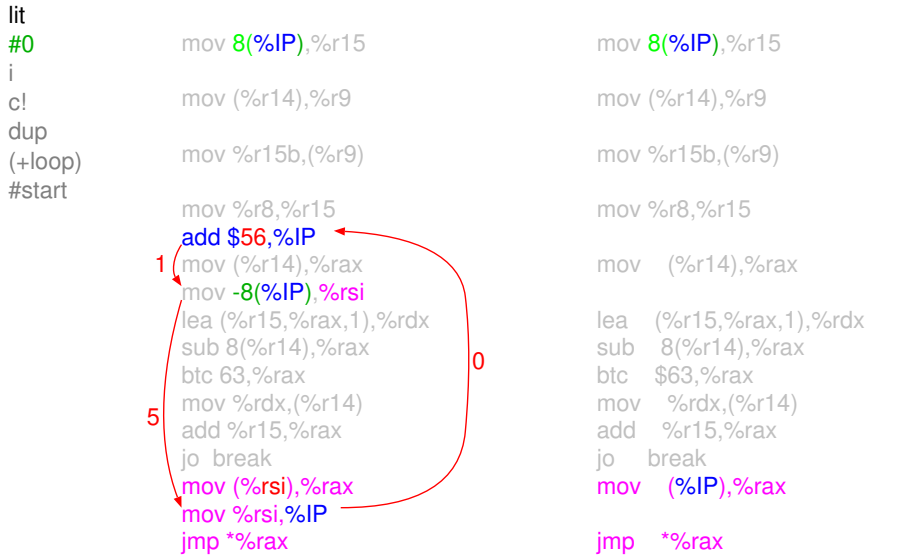
c: combine instruction pointer updates



ci: ... and optimize immediate VM instructions



cib: ... and optimize VM branch instructions



I: break loop dependencies

```
lit
#0
i
c!
dup
(+loop)
#start
```

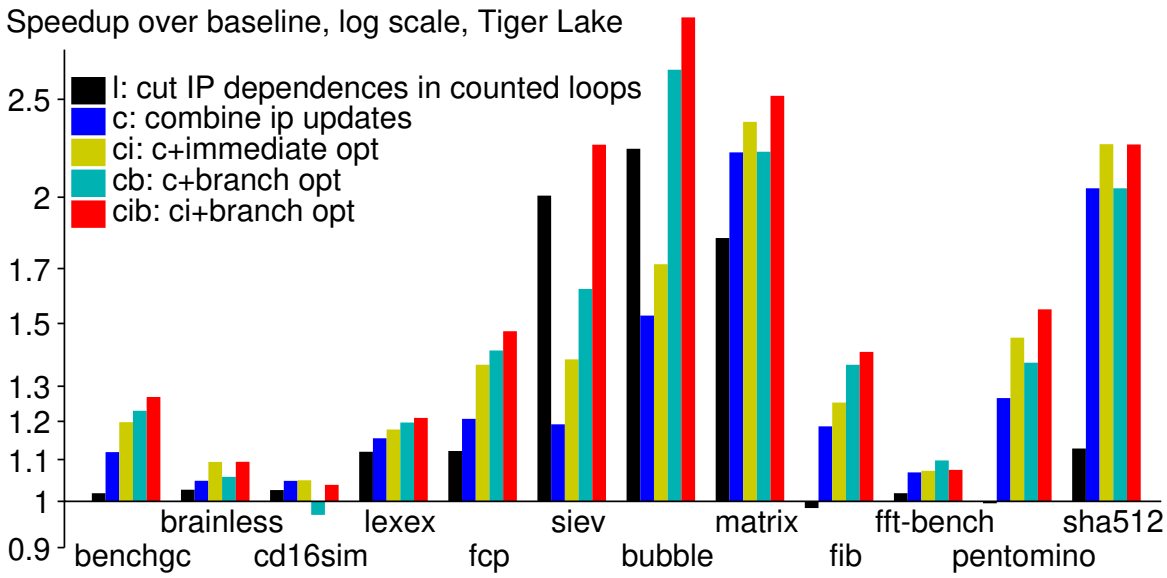
```
add $16,%IP
1 mov -8(%IP),%r15
add $8,%IP
1 mov (%RP),%r9
add $8,%IP
1 mov %r15b,(%r9)
add $8,%IP
1 mov %r8,%r15
add $16,%IP
1 mov (%RP),%rax
mov -8(%IP),%rsi
  lea (%r15,%rax,1),%rdx
  sub 8(%RP),%rax
  btc 63,%rax
  mov %rdx,(%RP)
5 add %r15,%rax
  jo  break
mov (%rsi),%rax
mov %rsi,%IP
jmp *%rax
```

0

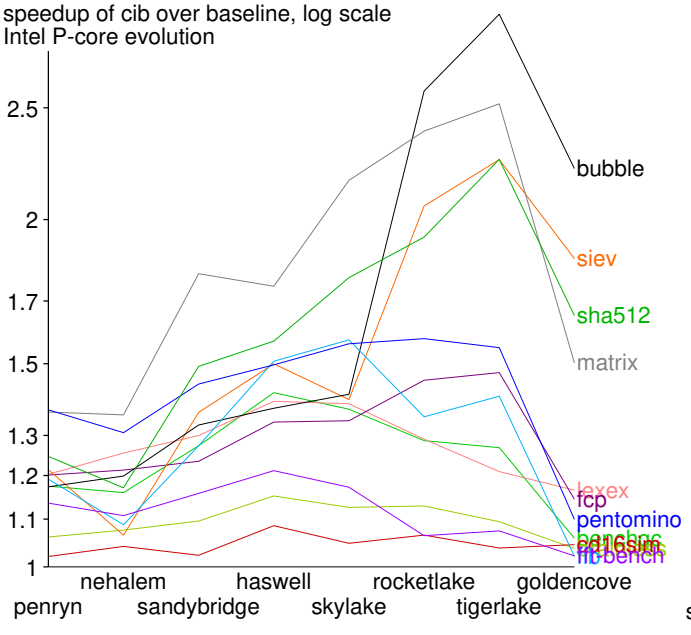
```
add $16,%IP
1 mov -8(%IP),%r15
add $8,%IP
1 mov (%RP),%r9
add $8,%IP
1 mov %r15b,(%r9)
add $8,%IP
1 mov %r8,%r15
add $8,%IP
  mov (%RP),%rax

  lea (%r15,%rax,1),%rdx
  sub 8(%RP),%rax
  btc 63,%rax
  mov %rdx,(%RP)
  add %r15,%rax
  jo  break
mov 16(%RP),%IP
mov 0(%IP),%rax
jmp *%rax
```

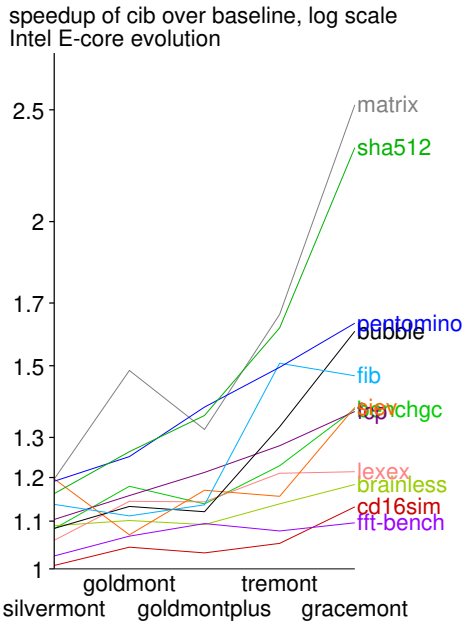
5

9



10





11

Conclusion

- Problem: VM instruction-pointer updates can be a performance bottleneck
- Solution: Optimize instruction-pointer updates
  combine them
  immediate operand variants
  branch to (adjusted) instruction pointer
  load **l**oop start address without using the instruction pointer
- Results
  speedup factors > 2 on loop-dominated benchmarks: critical path
  speedup factors 1.1–1.3 on call-dominated benchmarks
- Paper: DOI: 10.4230/LIPIcs.ECOOP.2024.14
  `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.14`

12

Gforth (interpreter) vs. SwiftForth (JIT)

Gforth VM	Gforth machine code	source	SwiftForth

lit
#0
	mov 0x8(%rbx),%r15	0	lea -0x8(%rbp),%rbp
			mov %rbx,0x0(%rbp)
			mov $0x0,%ebx

i	mov (%r14),%r9	i	lea -0x8(%rbp),%rbp
			mov %rbx,0x0(%rbp)
			mov %r14,%rbx
			add %r15,%rbx

c!	mov %r15b,(%r9)	c!	mov 0x0(%rbp),%eax
			mov %al,(%rbx)
			mov 0x8(%rbp),%rbx
			lea 0x10(%rbp),%rbp

dup	mov %r8,%r15	dup	lea -0x8(%rbp),%rbp mov %rbx,0x0(%rbp)

(+loop)
#start
	mov (%r14),%rax
	lea (%r15,%rax,1),%rdx
	sub 0x8(%r14),%rax
	btc $0x3f,%rax
	mov %rdx,(%r14)
	add %r15,%rax
	jo end
	mov (%rbx),%rax
	jmp *%rax
		+loop	add %rbx,%r14
			mov 0x0(%rbp),%rbx
			lea 0x8(%rbp),%rbp
			jno start

13

Why is gcc -O3 so slow for *bubble*?

```
    gcc -01                      gcc -03

1c: add   $0x4,%rax         c0: movq   (%rax),%xmm0
    cmp   %rsi,%rax             add    $0x1,%edx
    je    35                    pshufd $0xe5,%xmm0,%xmm1
25: mov   (%rax),%edx           movd   %xmm0,%edi
    mov   0x4(%rax),%ecx        movd   %xmm1,%ecx
    cmp   %ecx,%edx             cmp    %ecx,%edi
    jle   1c                    jle    e1
    mov   %ecx,(%rax)           pshufd $0xe1,%xmm0,%xmm0
    mov   %edx,0x4(%rax)        movq   %xmm0,(%rax)
    jmp   1c                e1: add    $0x4,%rax
35:                             cmp    %r8d,%edx
                                jl     c0
```

14