39th EuroForth Conference

September 15 – 17, 2023

Hotel Villa Aricia
Rome
Italy

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 39th EuroForth finds us in Rome where we finally meet again in person, after three years of online meetings. 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 no submissions to the refereed track. Nevertheless, I want to thank the program committee for their willingness to review papers.

In addition to the papers available before the conference, these online proceedings also contain 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/ef23/papers/`.

Workshops and social events complement the program. This year's EuroForth has been organized by Gerald Wodni.

Anton Ertl

Program committee

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

Contents

Prospective values and Forth

Bill Stoddart, Frank Zeyda

October 12, 2023

Abstract

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 sequential programming language that incorporates backtracking and speculative computations. Here we try it out with Forth.

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], and shown that it provides a sufficient formalism for describing backtracking and reversible computations. Our theory is developed as an extension of the B-Method. [Abr96]. 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]. One motivation for providing a PV semantics for Forth itself would be to provide a means of checking the validity of the code produced by such a compiler.

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 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.

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

SP! $\diamond$ $s = \varepsilon$
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 $[\![\mathrm{E}]\!]$ to represent this translation. Some examples will make this clear.

[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.

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

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

Next consider the translation of a simple expression:

$[\![\text{x 10 +}]\!] = x + 10$

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

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 caused 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:

$x\ 10\ +\ \mathit{to}\ x\ \diamond\ 2*x+y$ = 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.
$$\begin{array}{l} \mathit{top}(s)\ \widehat{=}\ R(s) \\ \mathit{next}(s)\ \widehat{=}\ R(L(s)) \end{array}$$

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:

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

and so on

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

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

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 $\quad \mathrm{S\ T}\ \diamond\ E\ =\ \mathrm{S}\ \diamond\ \mathrm{T}\ \diamond\ E$

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

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

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 $\diamond\ (\lambda\, s \bullet s)\mathit{drop}(s)\ =$ by lambda evaluation
SWAP $\diamond\ \mathit{drop}(s)\ =$ by semantics of SWAP
$(\lambda\, s \bullet \mathit{drop}(s))\mathit{swap}(s)\ =$ by lambda evaluation
$\mathit{drop}(\mathit{swap}(s))\ =$ applying *swap*
$\mathit{drop}(L^2(s) \mapsto \mathit{top}(s) \mapsto \mathit{next}(s))\ =$ applying *drop*
$L^2(s) \mapsto \mathit{top}(s)\ =$ 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 intermediate result step $\mathit{drop}(\mathit{swap}(s))$ 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 guarded no-op removes the flag and execution continues ahead. If g leaves a false flag, its 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.

$\mathit{null}\ =\ \sim\{\ \}$

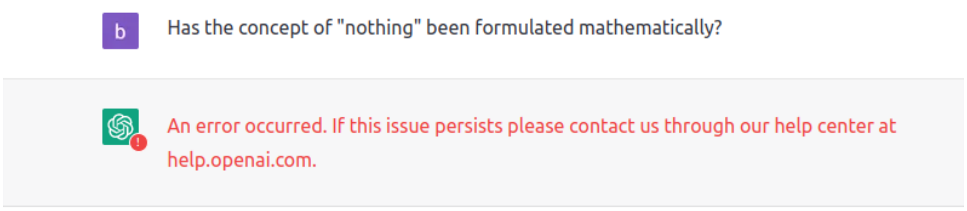
Bunch union has the properties: $S, T\ =\ T, S$ and $S, \mathit{null}\ =\ S$, and an additional property of *null* is $\{\ \mathit{null}\ \}\ =\ \{\ \}$.

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

$$\mathit{true}\ \multimap\ E\ =\ E,\ \ \mathit{false}\ \multimap E\ =\ \mathit{null}$$

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

Admittedly, this is pretty weird till you get used to it, and these concepts are not widely known. When we asked chatGPT about "nothing" we got the following response:


b Has the concept of "nothing" been formulated mathematically?

An error occurred. If this issue persists please contact us through our help center at help.openai.com.

Our semantic rule for guard is

g –> $\diamond E \;=\; g \rightarrow E$

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.

$[\![$ 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, and if not forces backtracking. Computation returns to the previous choice and continues ahead once more with the unused choice being selected. this time the guard lets computation continue ahead, with x=2. 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> $\diamond\, x = 2 \multimap x$ $=$ by semantics of choice

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

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

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

2.

7.1 Conditionals

We can think of 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:

IF S ELSE T THEN $\diamond\, E \;=\; (top(s) \neq 0 \;\multimap\; \text{S} \,\diamond\, E)\;,\;(top(s) = 0 \;\multimap\; \text{T} \,\diamond\, E)$

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, restoring 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> $\diamond\, s \;=\; s \mapsto (\text{S} \,\diamond\, E)$

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:

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

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 fom 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> 4 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 $\surd n$ to represent the perfect integer square root of n, it is clear for example that no integer satisfies $\surd 2$, and we capture this in our theory by saying $\surd 2 = null$. 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 signature INT POW.

```
: 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
```

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, 7FFF 1 + violates the 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:

P | 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 nondeterminism 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 for 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

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:

$V > 0 \wedge$ 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 $\diamondsuit$ $V = top(s)$ 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 Conclusions

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 operations, 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 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.

The mathematical expression of nothing as the constant *null* plays a key role in our semantics. We also 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 computational interpretation being that such an application triggers backtracking.

We have shown how prospective value semantics provides a description of stack based operations, but a full description of Forth semantics, covering interpretation and compilation, memory access, and the definition of defining words, is beyond the theory presented here. Nevertheless we can extend this investigation to provide a semantics that is *usable) for developing Forth applications, and we hope to report on the best way to do this in our future work.*

References

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

[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. 2023 edition available on-line.*

[RS10] *C Ritchie and W J Stoddart. A compiler which creates type tagged parse trees and executes them as Forth programs. In* 26th EuroForth Conference Proceedings*, 2010.*

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

On Solving Hexadoku and Debugging Recursive Programs with Message Digests of the Data Stack

François Laagel*
Institute of Electrical and Electronics Engineers

Abstract

Debugging Forth code is difficult, especially when dealing with recursive code. One has to maintain invariants and ensure that the data stack is not unduly modified. This document presents a new technique called *stack digesting* which helps the programmer quickly converge on coding errors. It is based on a cryptographic message digest algorithm that is completely specified in [1]. Although the NIST deprecated this message digest generation mechanism in 2011, it serves our debugging purposes well enough. The idea is based on the insertion of strategically placed probing points in the code being debugged so as to make sure that invariants are actually preserved.

1 Problem Statement

Recursion often is the most natural way to express an algorithm when dealing with intrinsically combinatorial problems (see [2]). Elektor Magazine is a publication that caters to electronics enthusiasts of all kinds. Every two months they publish an issue which includes a puzzle called *Hexadoku*. The rules are similar to the traditional Sudoku puzzle. However Hexadoku extends the regular 3 by 3 nature of Sudoku to a 4 by 4 use case. As a result we end up with a 16 by 16 grid instead of the traditional 9 by 9. Each spot in the grid can be any digit expressed in hexadecimal. The usual Sudoku integrity constraints apply; they are simply extended:

- digit uniqueness in a 4 by 4 sub-quadrant.
- digit uniqueness in an horizontal row.
- digit uniqueness in a vertical column.

This kind of puzzle begs for an automated problem solver and one way to approach it is to think of the problem in a three dimensional manner. The grid itself is implemented as a two dimension array of cells. Each cell of the array is interpreted either as a known value (a power of two corresponding to a resolved value) or a bitmask corresponding to the sum of viable alternatives for that spot. This is similar to the notion of "sum over all possibilities", a central tenet of the path integral formulation of quantum theory (see [3]).

The automated solver I implemented works not by looking for a solution but by converging on one by systematic elimination of unviable alternatives. It does so by alternating between phases of inference and speculation until eventually all spots are resolved (have a cell value that is a power of two).

- `infer` goes systematically over the whole grid and updates bitmasks based on resolved spots values, taking into account the constraints defined by the rules of the puzzle. In effect, this greatly reduces the size of the search space.
- `speculate` selects a currently unresolved spot location and recursively explores the space of open possibilities for that spot. It uses feedback supplied by `infer` to detect constraints violations and backtrack when appropriate. An application specific *transaction stack* is used to record all inferred changes to the grid and to undo them when constraints are detected as being violated. A transaction boundary occurs (and is flagged as such) whenever a speculative decision is made. Basically a transaction includes all inferred changes to the grid since the latest speculation.

In essence it works as a greedy minimalization algorithm aiming at reducing the number of unresolved spots values to zero. Its kernel is as outlined in figure 1. Some implementation details need to be elaborated on at this point.

- `rl+`/`rl-` increment and decrement a global variable that holds the current recursion level. The maximum recursion level also is maintained.
- `get-unresolved` selects the first unresolved grid cell for which the number of set bits is minimal but stricly greater than one. This is where the greedy aspect of the algorithm originates from since we always attack the problem from an angle where the number of options is the smallest at any given point in time.

*`f.laagel@ieee.org`

```
: speculate ( -- success-flag )
  rl+                           \ Increment recursion level

  get-unresolved                \ Look for an unresolved spot
  DUP 0= IF INVERT EXIT THEN    \ Problem solved

  DUP @                         \ S: saddr\sval
  \ The list of set bits in TOS indicate the possibilities
  \ for the selected spot. Explore these alternatives.
  16 0 DO
    DUP I 2^n AND IF
      OVER TRUE SWAP tstk-push \ Insert transaction boundary

      OVER I 2^n SWAP
        +ul |visual -ul !       \ Un-logged update-spot

      infer IF                  \ No inconsistencies detected
        RECURSE IF              \ Stop on the 1st solution found
          2DROP UNLOOP TRUE EXIT
        THEN
      THEN

      \ Backtrack up to the last transaction boundary.
      BEGIN tstk-pop UNTIL
      nbt 1+!               \ Increment the number of backtracks

    THEN
  LOOP

  2DROP FALSE                   \ Dead end reached
  rl- ;                         \ Decrement recursion level
```

Figure 1: The Core Speculation Engine

- `tstk-push`/`tstk-pop` take care of handling the transaction stack. `speculate` goes over the alternatives for one selected spot only. Which explains the `16 0 DO ... LOOP` construct. Whenever a speculative decision is made, that change is logged to the the transaction stack under a *boundary* marker. Changes later inferred will also be stacked up so that that they can be undone, should the current speculative decision prove not to be conducive to a workable solution.

- `|visual` updates the on-screen representation of the current solution state. It is stack neutral but uses the *next of stack* as the new grid cell value for the spot pointed to by the *top of stack*.

- `+ul`/`-ul` select and deselect the underline font style in order to display the speculation spots in a way that stands out.

- `infer` will deduce all the consequences of a speculative decision until either a constraint violation is detected or the number of unresolved spots can no longer be decreased (a viable situation).

During the execution of the loop in `speculate`, it is essential that the top two values of the data stack be preserved. They are the cell address of the unresolved spot we are working on and that spot's original superposition of possible states.

I quickly devised a working implementation that ultimately converged on a solution matching the constraints defined by the rules of the game. Yet, it was not entirely satisfying since zeroes–a zero grid cell value indicates an impossible condition for the associated spot–started surfacing in the grid and I thought that this situation was not detected early enough, thereby negatively impacting the overall performance of the solver.

```
>speculate at rl 15
               [ 4 , 1 ] <- 5          Cell owned at rl 15
>speculate at rl 16
                [ 4 , 12 ] <- 7
>speculate at rl 17
                 [ 4 , 4 ] <- C
                 [ 4 , 7 ] <- E
                 [ 5 , 7 ] <- 4
                 [ 10 , 7 ] <- 4       Vertical constraint violation
                 [ 10 , 5 ] <- 8
                 >backtrack
                 [ 10 , 5 ] <-
                 [ 10 , 7 ] <-
                 [ 5 , 7 ] <-
                 [ 4 , 7 ] <-
                 [ 4 , 4 ] <-
                 <backtrack
<speculate at rl 17
                >backtrack
                [ 4 , 12 ] <-
                <backtrack
                [ 4 , 1 ] <- E         Cell contents altered at rl 16!
```

Figure 2: Failure at Recursion Level 16

So I worked on early detection and avoidance of zero grid cell values. Associated changes to the source code were mostly in the `infer` code. As it turned out, this is where things started going sideways. I had introduced a bug somewhere in the 650 lines or so of code of the solver.

In any sufficiently complex code, these things are bound to happen and I think it is no secret that the classic ways of handling such a conundrum are:

- systematic verification of assumptions. Preconditions can easily be verified by extra tests and references to `ABORT"`. Working code should already have this defense mechanism built-in but if it does not, this is definitely the first step to be taken toward fixing a bug.

- an efficient logging mechanism. This is priceless and also should be an integral part of the original code. Logging should be conditional based on some ad'hoc *debug* vector flag.

- last resort measures such as improved code documentation and/or third party code reviews. These are either time consuming, expensive or both– definitely not practical ways to address the problem at hand.

Post-condition verification is somehow more elusive and the object of this paper is precisely to describe one, with respect to data stack integrity preservation.

After having implemented a reasonable logging mechanism, I realized the code failed to preserve the key invariant `speculate` relies on. It did so at recursion level 16, as illustrated in figure 2. In order to fix this bug rapidly, I introduced the concept of *stack digests.*

```
>speculate at rl 15
               [ 4 , 1 ] <- 5
               >infer 390D7AFB:87768414:F88C2553:C3F8F2A4:4C74CE4F
               <infer 390D7AFB:87768414:F88C2553:C3F8F2A4:4C74CE4F
>speculate at rl 16
                [ 4 , 12 ] <- 7
                >infer 36BF7E72:0BFB296F:E13498C8:D6A8C5DF:C3B6C3D9
                <infer 36BF7E72:0BFB296F:E13498C8:D6A8C5DF:C3B6C3D9
>speculate at rl 17
                 [ 4 , 4 ] <- C
                 >infer F0812B19:93C42F48:2C610131:FAD0D5D1:24E108A3
                 [ 4 , 7 ] <- E
                 [ 5 , 7 ] <- 4
                 [ 10 , 7 ] <- 4
                 [ 10 , 5 ] <- 8
                 <infer 36BF7E72:0BFB296F:E13498C8:D6A8C5DF:C3B6C3D9
```

Figure 3: `infer` code is the culprit

2 Proposed Solution and Proof of Concept

The original concept was proposed on the *Forth2020* Facebook group and was well received by some of its most respected contributors. [4] proposed a stack checking mechanism that only covers stack depth changes. Later on, [5] developed a complete testing framework for ANS94 compliance. To this day, it remains a reference tool for most testers.

Stack digesting covers both depth and actual stack contents but it does not try to perform stack effect characterization at all. It is a debugging aid only meant to be used as an integrity checking tool.

Basically, a stack digest is just what its name suggests: a cryptographic message digest of the state of the data stack (or a subset of it) as it is when sampled or verified–in this paper support for automated verification is not addressed; visual inspection of the logging output is required for this concept to be of any use. The API is restricted to a single word:

> `SDIGEST` (i*x u – i*x) Prints a cryptographic digest of the contents of the data stack, omitting the topmost *u* cells. The algorithm used for producing this digest is implementation defined.

A quick survey of commonly off the shelf available hashing algorithms revealed those most generally agreed upon were crytographic signatures. Among them, it turned out that the easiest to implement and the best documented one was SHA1. [6] provides a detailed pseudo-code description for SHA1.

Once equipped with this new technology, we are now in a position to instrument the application code and to quickly converge on the problem's root cause (figure 3).

The inference code is about 200 lines long but its primary routine is `reduceall` (see figure 4). Through additional code instrumentation, it was determined that an extraneous `2DROP` reference in `reduce4x4` was responsible for this unwarranted data stack corruption.

```
: reduceall ( -- failure-flag )
  reduce4x4 IF                  \ Constraint violated
    TRUE EXIT
  THEN

  16 0 DO
    I get-horiz-mask IF         \ Constraint violated
      UNLOOP TRUE EXIT
    THEN
    ( S: new-possibly-zero-mask ) I SWAP set-horiz-mask IF
      UNLOOP TRUE EXIT
    THEN

    I get-vert-mask IF          \ Constraint violated
      UNLOOP TRUE EXIT
    THEN
    ( S: new-possibly-zero-mask ) I SWAP set-vert-mask IF
      UNLOOP TRUE EXIT
    THEN

  LOOP
  FALSE ;
```

Figure 4: The `reduceall` word

3 Status and Further Work

An actual implementation for `SDIGEST` and the underlying SHA1 message digest generation is available at [7]. It has been validated against well known test vectors on 64 and 32 bit cell targets, regardless of their endianness. This framework has been successfully used to fix a nasty bug in the inference code of my unpublished proprietary *Hexadoku* automated solver.

The concept could be further extended by having a dedicated digest stack and manually coded verification checkpoints. Each entry on the digest stack would be the output of the SHA1 message digest code (5 cells). This would most likely require some extension of the API and is left as an exercise to the interested reader.

4 Conclusion

This article was written in the hope that stack digesting could become a useful tool in the Forth programmer's toolbox. It provides a convenient synthetic overview of the state of the data stack, which can be very deep indeed.

Recursion is a powerful technique which allows the developer to formulate solutions to complex problems in very simple terms. However, its use is generally frowned upon in the context of embedded systems software development because stack utilization is basically unpredictable, especially when dealing with heavily data driven algorithms.

References

[1] D. Eastlake 3rd, P. Jones
RFC 3174: US Secure Hash Algorithm 1 (SHA1)
The Internet Society, September 2001.

`https://www.rfc-editor.org/rfc/rfc3174`

[2] Donald L. Kreher, Douglas R. Stinson
COMBINATORIAL ALGORITHMS Generation, Enumeration and Search
Chapter 4, Backtracking Algorithms
CRC Press, 2019.

[3] Markus Pössel
The sum over all possibilities: The path integral formulation of quantum theory
Einstein Online, 2006.
`https://www.einstein-online.info/en/spotlight/path_integrals/`

[4] Ulrich Hoffmann
Stack checking - A debugging aid
euroFORML Conference Proceedings, 1991.
`http://www.euroforth.org/ef91/hoffmann.pdf`

[5] John Hayes S1I
Core ANS94 Test Harness
Online contents, November 27, 1995.
`http://www.forth200x.org/tests/ttester.fs`

[6] National Security Agency (original designers)
SHA-1
Wikipedia.org, various contributors.
`https://en.wikipedia.org/wiki/SHA-1`

[7] François Laagel
SHA-1 sample code for GNU Forth 0.7.3 or SwiftForth 3.7.9
Online contents, August 15, 2023.
`https://github.com/forth2020/frenchie68/blob/main/sdigest-generic.4th`

EuroForth 2023

Accessing an Oracle database using Forth

Abstract

It is occasionally necessary to access an Oracle database from a Forth applications. This paper illustrates the techniques we have used.

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

Within our own automation applications, we use the MySQL database. In previous papers, I have described the method that we use for accessing a MySQL database from Forth, which we refer to as "Forth Query Language" or FQL.

Despite the many benefits of MySQL, there are some of our customers who insist on using an Oracle database for their office systems.

It is therefore necessary for us sometimes to access an Oracle database, for the exchange of data about such things as category information and machine efficiency.

The technique required for Oracle is quite different from that required for MySQL.

2. A brief comparison of Oracle and MySQL

	Oracle	**MySQL**
Cost	Very expensive	Free
License	Proprietary	GPL
Paid support	Yes	Available
Free support	None	Strong
Performance	4.3/5	4.4/5
Indexing	Full text, hashed, binary	Full text, hashed
Learning curve	Very steep	Modest
Almost everything else	Same	Same

3. The available Oracle APIs

For many years, there were only two possible methods for accessing an Oracle database - the "Oracle Call Interface", and ODBC.

For anyone accustomed to the "C" API for MySQL, the Oracle Call Interface feels extremely complex and very hard to master.

ODBC adds an additional layer to the interface, and in the past we have experienced problems with versioning compatibilities of libraries and drivers.

Recently, a new API has been introduced - "Oracle Database Programming Interface for C" (ODPI-C). This is essentially a simplified wrapper that sits on top of the Oracle Call Interface. It is still more complex that the MySQL interface, but much more manageable than before. This is therefore the method we have chosen.

4. Typical tasks that are required

The actual database tables are not normally exposed on an Oracle system. To retrieve data from Oracle, one would normally use a SELECT statement, applied to a specially constructed "view". To insert data into Oracle, typically a prepared statement is called. The prepared statement would carefully constrain the provided parameters, in order to stop anything nasty from being inserted.

5. Declaring the necessary library function

Given the restricted types of operations normally required, only a small subset of the available ODPI-C functions needs to be declared in Forth, in just four different classes. This is still more than the number of functions needed for a comprehensive MySQL interface, but not too onerous.

For example, in the "statement" class, just six functions are needed.

```
\ DPISTMT

extern: int dpiStmt_execute( dpiStmt * stmt, dpiExecMode mode, uint32_t *
                             numQueryColumns );
extern: int dpiStmt_getRowCount( dpiStmt * stmt, uint64_t * count );
extern: int dpiStmt_fetch( dpiStmt * stmt, int * found, uint32_t * bufferRowIndex );
extern: int dpiStmt_defineValue( dpiStmt * stmt, uint32_t pos,
                                 piOracleTypeNum oracleTypeNum,
                                 dpiNativeTypeNum nativeTypeNum,
                                 uint32_t size, int sizeIsBytes,
                                 dpiObjectType * objType );
extern: int dpiStmt_getQueryValue( dpiStmt * stmt, uint32_t pos,
                                   dpiNativeTypeNum * nativeTypeNum,
                                   dpiData * *data );
extern: int dpiStmt_release( dpiStmt * stmt );
```

It will be seen that there are a large number of bespoke data types, and in order to ensure the accuracy of the external declarations, I always define these explicitly in the usual way, for example:

```
also types definitions
: dpiContext void ;
: dpiCommonCreateParams void ;
: dpiContextCreateParams void ;
: dpiConn int ;
: dpiErrorInfo int ;
...
previous definitions
```

A small number of structures also need to be replicated in Forth, and these need to be carefully checked for offset values against the equivalent "C" structures, so as to avoid any possible alignment problems. A typical structure is:

```
STRUCT dpiErrorInfo
  INT             dei.code
  INT             dei.offset16
  POINTER         dei.message
  INT             dei.messageLength
  POINTER         dei.encoding
  POINTER         dei.fnName
  POINTER         dei.action
  POINTER         dei.sqlState
  INT             dei.isRecoverable
  INT             dei.isWarning
  INT             dei.offset
END-STRUCT
```

Note that on a 64 bit system, an INT is still 32 bits!

Finally a selection of constants is required, and in this case they can be copied directly from the "C" header file and into Forth, for example:

```
\ Native type numbers

#define DPI_NATIVE_TYPE_INT64                       3000
#define DPI_NATIVE_TYPE_UINT64                      3001
#define DPI_NATIVE_TYPE_FLOAT                       3002
#define DPI_NATIVE_TYPE_DOUBLE                      3003
#define DPI_NATIVE_TYPE_BYTES                       3004
#define DPI_NATIVE_TYPE_TIMESTAMP                   3005
...
```

6. A Forth wrapper for an Oracle query

We first made some design decisions in order to simplify the wrapper.

a) Our interface does not need to be thread safe. Only one thread will be used to access the Oracle system. Therefore all data such as handles of connections or statements can he held in Forth VALUEs. If a thread safe interface were required, these could be moved into Forth user variables.

b) Because we query the Oracle database relatively infrequently (e.g. once every 10s), and because individual connections do not appear to be very costly in terms of processing time, we opted to create a new connection for each query. This greatly simplifies the error handling.

c) The Oracle server PC and its connections are not under our control. The server may occasionally be down for maintenance. If that happens, it is important that no data is lost. We therefore maintain a queue of data to be sent to Oracle, and offer the queries one by one with automatic retry, including backoff.

```
: ORACLE-QUERY { pzquery -- f } \ True if query prepared and executed
  FALSE                                          \ Assume failed
  ORACLE-INITCONTEXT IF                          \ Initialise Oracle context
    ORACLE-RELEASE                               \ Release any previous statement
    ORACLE-DISCONNECT                            \ New connection for each query
    ORACLE-CONNECT IF                            \ Connected OK
      pzquery ORACLE-PREPARE IF                  \ Statement prepared OK
        ORACLE-EXECUTE IF                        \ Statement executed OK
        ORACLEMINRETRYTIME -> ORACLERETRYTIME    \ Set error retry time to minimum
          DROP TRUE                              \ Success
        THEN
      THEN
    THEN
  THEN
  DUP IF                                         \ Success
    ORACLEMINRETRYTIME -> ORACLERETRYTIME        \ Set error retry time to minimum
  ELSE                                           \ Failure
    ORACLECREATEDCONTEXT dpiContext_destroy DROP \ Destroy context
    0 -> ORACLECONTEXT                           \ Clear context
    ORACLE-RETRYWAIT                             \ Wait before retry
  THEN
;
```

The first step is to check for a connection "context", re-creating one if necessary. Then we ensure that everything is tidy after any previous query. The connection is then established.

Unlike in MySQL, query statements in Oracle have to be explicitly prepared first before being submitted for execution.

If all the above steps are successful, we can exit with a true flag, and as we go, reset the backoff time setting.

In the event of failure, we destroy the connection context (if the failure was due to some change in the configuration of the Oracle server, then a new context will be required). We then leave a gradually increasing amount of time before trying again.

We can look at one of those steps in slightly more detail, for example:

```
: ORACLE-ERROR ( ---zperror ) \ Returns an Oracle error message
  ORACLECONTEXT ORACLEERRORINFO dpiContext_getError   \ Populate error info struct
  ORACLEERRORINFO dei.message @                       \ Return error message
;

: ORACLE-CONNECT ( ---f ) \ True if Oracle connection established
  ORACLECONTEXT ORACLE-USER ORACLE-PASSWORD ORACLE-DATABASE NULL NULL
  ADDR ORACLECONNECTION dpiConn_create DPI_FAILURE = IF
    Z" Failed to connect to Oracle, " ORACLE-ERROR Z+ ERROR
    FALSE
  ELSE
    TRUE
  THEN
;
```

The connection parameters are normally regarded as valuable information, so need to be kept somewhere secure.

Note the recovery of the address of the full text of an error, from the error information structure as shown earlier. This is then added to our own log file, so we can see in detail about anything that goes wrong.

7. A typical SQL INSERT query

Because the insert queries are normally quite short and straightforward, we have not attempted to replicate the FQL system for constructing queries. Instead, the query is simply put together using the zero-terminated string concatenator Z+, which uses a scratchpad formed by dividing PAD into three sections. This is fine when the query can be guaranteed to be shorter than /PAD 3 / but would be no use in the MySQL general case, where queries can be very long.

```
: EXAMPLE-EXPORTDATA ( ---f ) \ Send data to Oracle
  Z" CALL EXPORT_MICROSS("                \ EXPORT_MICROSS cust, …, system
  ZQ Z+ EXPORTCUSID Z+ ZQ Z+ ZCOMMA Z+    \ Customer ID
  ...
  EXPORTAREA ZFORMAT Z+                   \ Sorting area
  Z" )" Z+
  ORACLE-QUERY
;
```

Note that the SQL INSERT statement is not used directly, instead a statement is prepared by the Oracle database supervisor, which we are able to use through an SQL CALL statement.

8. A typical SQL SELECT query

Here, the SQL SELECT statement is used directly, but is applied to a view rather than to a table directly. The view is declared by the Oracle database supervisor and contains the subset of the table columns that we are permitted to see.

```
: IMPORTPRODUCTS { | pcusid[ LENCUSID 1+ ] pcatid[ LENCATID 1+ ] -- }
\ Import products from Oracle
  FALSE                                                 \ Assume fail
  Z" SELECT CUSTOMERID, PRODUCTID "
  Z" FROM IMPORTPRODUCTS" Z+
  ORACLE-QUERY IF
    BEGIN
      ORACLE-FETCH
    WHILE
      1 ORACLE-NUMBER ZFORMAT ZCOUNT pcusid[ LENCUSID ZPUT  \ Get customer ID
      2 ORACLE-STRING                pcatid[ LENCATID ZPUT  \ Get category ID
      ...
    REPEAT
    DROP TRUE                                           \ Success
  THEN
;
```

Having run the query, we now need to analyse the results. In Oracle, results are provided in a completely different way from MySQL.

Let us first look at our wrapper word that fetches one row of results.

```
: ORACLE-FETCH { | pfound -- f } \ True if row of query result fetched
  0 -> pfound
  ORACLESTATEMENT ADDR pfound ADDR ORACLEINDEX dpiStmt_fetch
  DPI_SUCCESS = pfound 0<> AND
;
```

It is worth looking at this carefully, because it illustrates a frequent source of bugs when using Forth.

Like all "C" functions, dpiStmt_fetch can return only one parameter, and this is always a code to indicate if the function worked or not (e.g. DPI_SUCCESS). This function however also needs to tells us if it actually returned a row of data or not. We therefore need to provide as a parameter an address of where to put that information. This information is needed only within the Forth word, so we use a local value "pfound" and pass ADDR pfound to our "C" function.

Assuming that dpiStmt_fetch executes OK, then it will have written either true or false into "pfound".

But here is where the problem lies. We are in 64 bit Forth, therefore "pfound" is a 64 bit value. But dpiStmt_fetch writes a boolean value, which in 64 bit Linux is represented by an INT, which is a 32 bit value. Forth local values are not initialised, so at the start of the function "pfound" is random. After the "C" function returns, "pfound" still returns nonsense.

It is essential to remember (but very easy to forget) that local values like these must be explicitly initialised in the code "0 -> pfound".

This is such a common source of errors, that I would propose that the Forth standard should be changed to require that local values are automatically zeroed.

9. Analysing the Oracle result set

In MySQL, there is only one data type in a result set. The set consists solely of an array of pointers to zero terminated strings.

In Oracle, it is much more complicated because each column data type returns a different structure, and the result set consists of pointers to structures.

```
: ORACLE-STRING { pcol | pbytes -- p$ plen } \ Get string value at column
  ORACLESTATEMENT pcol ADDR ORACLETYPE ADDR ORACLEPDATA dpiStmt_getQueryValue DROP
  ORACLEPDATA dpiData_getBytes -> pbytes
  pbytes dpiBytes.ptr @ pbytes dpiBytes.length I@
;
```

In the example above, we recover a string result. This is a three stage process.

First we have to get the data type, and a pointer to the data. For simplicity, we have not included any error checking here - we assume we have read the customer's specification of the view correctly.

Then, from the data pointer, we extract a pointer to the string structure.

Finally, from the string structure, we extract the base address and length of the string. Note that though when tested we find that the string is in fact zero terminated, the documentation does not actually say so, therefore, we need to assume that it isn't.

10. Conclusion and future work

This set of wrapper functions make it reasonably easy to make simple queries to an Oracle database. If more complex queries are needed, it would be possible to extend the FQL system to accommodate Orace as well as MySQL. A further development would be to make the word set thread safe.

EuroForth 2023

A proposed standard Forth style enumeration word set, using recognisers

Abstract

The lack of an ENUM word in the Forth standard has been previously noted.
The paper describes a solution that is fully faithful to Forth styling.
The implementation uses recognisers. Some potentially useful extensions will be discussed.

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

An enumeration is a data type which consists of a set of named members. Each member acts like a constant integer. A variable or value of the enumeration type can hold only integers within the named set. Enumerations are very useful, firstly for the elimination of "magic" numbers in code, and secondly for alerting any attempt to store an invalid number into an enumerated variable.

ANSI Forth does not include an enumeration word set.

The VFX implementation of Forth includes two enumeration words, but these are intended primarily for the import of "C" header files.

2. The existing VFX implementation

The two VFX enumeration words have the following form:

ENUM <enumname> {
  <membername>[=<integer>], [// <comment>]
  ...
};

For example:

```
ENUM GSignalMatchType {
  G_SIGNAL_MATCH_ID         = 1,      // The signal id must be equal
  G_SIGNAL_MATCH_DETAIL     = 2,      // The signal detail must be equal
  ...
  G_SIGNAL_MATCH_UNBLOCKED = 32,    // Only unblocked signals may be matched
                 };
```

There is also a similar type without an enumeration name:

ENUM{
  <membername>[=<integer>], [// <comment>]
  ...
};

These enable code to be copied directly from a "C" header file and pasted into Forth code. They work reasonably well, though a certain amount of adjustment if often required. For example, in the original header file, the above example looked like:

```
/**
 * GSignalMatchType:
 * @G_SIGNAL_MATCH_ID: The signal id must be equal.
 * @G_SIGNAL_MATCH_DETAIL: The signal detail must be equal.
 ...
 * @G_SIGNAL_MATCH_UNBLOCKED: Only unblocked signals may be matched.

...

typedef enum
{
  G_SIGNAL_MATCH_ID        = 1 << 0,
  G_SIGNAL_MATCH_DETAIL    = 1 << 1,
  ...
  G_SIGNAL_MATCH_UNBLOCKED = 1 << 5
} GsignalMatchType;
```

But, even after adjustment, this does not look like Forth!
a) comma separators
b) equals sign assignment
c) right-to-left assignment
d) "C" style comments

This is not a problem when they are used as intended, for imported code.
But when creating one own enumerations, this is a distraction. We use enumerations extensively, and there are approximately 60 different types in the code of our main application.

When an enumeration name is supplied, this cannot be used in subsequent code. The name is placed in a dedicated dictionary. Its only purpose is so that all the enumerations can be listed, using the word .NAMEDENUMS.

3. A proposed enumeration in the Forth style

We aimed for a solution that is as concise and easy to type as possible:

```
ENUM<< <enumname>
  [<Forth expression>] <membername> [\ <comment>]
  ...
>>
```

The previous example then becomes:

```
ENUM<< GSignalMatchType
  1 0 LSHIFT G_SIGNAL_MATCH_ID            \ The signal id must be equal
  1 1 LSHIFT G_SIGNAL_MATCH_DETAIL        \ The signal detail must be equal
  ...
  1 5 LSHIFT G_SIGNAL_MATCH_UNBLOCKED     \ Only unblocked signals may be matched
 >>
```

Now we look more Forth like:

a) white space separators
b) no equals sign
c) left-to-right assignment
d) Forth style comments
e) any Forth expression may be used, thus the original intention can be made clear

4. Implementation idea

It will be seen in the example that all Forth words between enumerator name and the terminator >> need to function exactly as normal, in interpretation mode. Only the new member names need to be dealt with, by creating new words that act like constants.

In effect, we need to "recognise", and process, only the new words.

This led to the idea of creating a new recogniser, which acts in effect as an "unrecogniser".

5. Understanding recognisers

It is not clear how recognisers suddenly turned into recognizers, but it seems we have to put up with that.

The first description of recognisers at a Euroforth conference was by Bernd Paysan in 2012. They became the standard technique for text interpretation in VFX Forth as from version 5.1

The first thing to notice is which recognizers are normally at work. In our case, using the SSE64 floating point package, these are:

```
.recognizers REC-FIND  REC-VOCDOT  REC-NUM  REC-SSEFLOATS   ok
```

The text interpreter offers a parsed word to each of the recognisers in turn, until one of them accepts it. Any word that is not accepted by any of the recognisers throws an undefined word error.

So, in the list above, a word is offered first to REC-FIND which deals with predefined Forth words, then REC-VOCDOT which deals with words in a specified vocabulary, then REC-NUM which deals with single and double length integers, and finally REC-SSEFLOATS which deals with 64 bit floating point numbers.

Each recognisers has a set of three possible actions, one for interpretation, one for compilation and one for postponement.

The key feature of recognisers is that they can be dynamically attached and detached during compilation.

6. Starting and ending enumeration

What is needed therefore, is for ENUM<< and >> to attach and detach respectively, a recogniser that deals with the enumeration members.

```
: ENUM<< ( <name>--- ) \ Start an enumeration
  ?EXEC                                   \ Only when interpreting
  ['] REC-ENUM FORTH-RECOGNIZER +STACK-BOT \ Add enumeration to the recogniser stack
  0 -> ENUMVAL                            \ Initialise enumeration value
  PARSE-NAME ($CREATE)                    \ Name of the enumeration
  HERE -> ENUMLIST                        \ Set root address of list
  0 ,                                     \ Initialise list
  ['] ENUMCOMP, SET-COMPILER              \ When an enumeration is being compiled
  INTERP> ENUMINTERP                      \ Interpret action
;

: >> ( --- ) \ End an enumeration
  ['] REC-ENUM FORTH-RECOGNIZER -STACK     \ Remove from the recogniser stack
;
```

Looking at the above line by line:

a) We can clearly only define an enumeration while interpreting
b) In VFX, recognisers are handled using a standard stack mechanism
c) Our enumeration recogniser goes at the bottom of the stack, to be dealt with last
d) The value of enumeration members starts by default at zero
e) All enumerations should be named
f) We make provision for a list of the enumeration members
g) We make provision for special compiling and interpreting actions when using a child of ENUM<< i.e. an enumeration name.

7. The enumeration recogniser

```
: REC-ENUM ( ??,caddr,u---??,caddr,u,r:??? ) \ The enumeration recogniser
  DEPTH 2 3 WITHIN IF   \ Zero or one new enumeration values defined
    R:ENUM
  ELSE
    CR ." Error defining new enumeration value"
    R:FAIL
  THEN
;
```

Each recogniser receives the parsed word as a caddr,u pair. It returns the handle of a recogniser structure, preceded by any necessary parameters. This is an opportunity to check for the validity of the optional Forth expression (if any). The only restriction on this expression is that it must result in either zero or one item on the number stack.

8. The enumeration recogniser action

```
: ENUMINTERPACTION ( ??,caddr,u--- ) \ Interpreter action for enumeration recogniser
  DEPTH 3 = IF                      \ A new enumeration value defined
    ROT -> ENUMVAL                  \ Set it
  THEN
  ($CREATE)                         \ Create the enumerated name
  ENUMVAL ,                         \ Set the constant value
  INC ENUMVAL                       \ Next enumeration number
  LATEST-XT ENUMLIST ATEXECCHAIN    \ Add to list
  ['] ENUMVALCOMP, SET-COMPILER     \ When enumerated constant is being compiled
  INTERP> ENUMVALINTERP             \ When enumerated constant is being interpreted
;

' ENUMINTERPACTION  ' NOOP  ' NOOP  RECTYPE: R:ENUM   ( ---struct )
\ Contains the three recogniser actions for enumeration
```

Only one action is required in this case, for interpret.

Looking at the above line by line:

a) If an optional Forth expression exists, and it results in one item on the number stack, then that number becomes the value for the next enumerated member
b) The member is then created. and the value is set
c) The value is then incremented ready for the next member
d) The member is added to the list of members for that enumerations
e) We make provision for special compiling and interpreting actions when using a child of ENUMACTION i.e. an member name

The interpret and compilation actions are, for the time being, simply those of a constant i.e.

```
: ENUMVALCOMP, ( xt--- ) \ Compiling action of an enumeration value
  >BODY @ CLIT,
;

: ENUMVALINTERP ( addr--- ) \ Interpret action of an enumeration value
  @
;
```

9. Showing the list of enumeration members

Initially, I simply defined the interpret and compilation actions of an enumeration name as "do nothing" - they just return the address of the member list.

```
: ENUMCOMP, ( xt--- ) \ Compiling action of an enumeration
  >BODY CLIT,
;

: ENUMINTERP ( addr--- ) \ Interpret action of an enumeration
;
```

This means that the members of the enumeration can be easily shown, as in the example below:

```
ENUM<< TESTENUM         \ Name of the enumeration
            AZERO       \ By default, the enumeration starts at zero
            AONE        \ Standard Forth comments are allowed
  1 2 +     ATHREE      \ Any Forth expression can be used to set the enumeration
            AFOUR       \ The enumeration increments
>>                      \ Enumeration terminator

TESTENUM SHOWCHAIN
AFOUR
ATHREE
AONE
AZERO  ok
```

10. More useful ideas - to do

One possibility might be to make the enumeration name create a value that is only allowed to hold the member numbers, for example:

```
TESTENUM MYENUMVAL
1 -> MYENUMVAL ok
2 -> MYENUMVAL
Invalid member 2 for enumeration MYENUMVAL ok
```

The list could still be shown by adding a modifier, such as:

```
MEMBERSOF TESTENUM
AFOUR
ATHREE
AONE
AZERO  ok
```

Another modifier might check if a number is a member of the enumeration, e.g.

```
1 MEMBEROF TESTENUM . 1 ok
2 MEMBEROF TESTENUM . 0 ok
```

Alternatively, the enumeration name could specify the field of a STRUCT, e.g.

```
STRUCT MYSTRUCT
  VINT            my1
  VWORD           myword
  VBYTE           mybyte
  100  VFIELD     mystring
  TESTENUM        myenum
END-STRUCT
```

where the myenum field was only allowed to hold the members of TESTENUM.

11. Conclusion

The use of recognisers makes it easy to create a Forth friendly enumeration word set. The use of modifiers allows an enumeration name to carry out a variety of functions.

Fix Spectre in Hardware!
Why and How

M. Anton Ertl[*]
TU Wien

Abstract

Spectre can be fixed in hardware by treating speculative microarchitectural state in the same way as speculative architectural state: On misspeculation throw away all the speculatively-performed changes. The resource-contention side channel needs to be closed, too. This position paper also explains how Spectre works, why software mitigations are not sufficient.

1 Introduction

Spectre [SSLG18] is a hardware vulnerability that has been reported to hardware manufacturers such as AMD and Intel in June 2017, and to the general public on January 3, 2018. Unlike Meltdown, which has been published at the same time and has been fixed in hardware relatively quickly[1] (or, in the case of AMD, not built into the hardware from the start), even the latest CPUs with speculative execution from all manufacturers are vulnerable to Spectre, and hardware manufacturers leave it to software to "mitigate" these vulnerabilities.

New Spectre variations that bypass existing mitigations are discovered regularly, e.g., the recent discoveries of Intel's DownFall [Mog23] and AMD's Inception [TWR23] vulnerabilities. Intel lists[2] 6 "transient execution attacks" published in 2018–2021, and, as of this writing (August 2023), 5 published in 2022-2023 that require software mitigations (sometimes with hardware support) even on Intel's most recent Sapphire Rapids server CPU. This includes the original two Spectre variants (v1 and v2) reported to Intel in June 2017.

In this position paper I present a general approach to fix Spectre in hardware (Section 9) that would fix not only Spectre v1 and v2, but also, e.g., the recently-discovered Downfall and Inception vulnerabilities. In order to make this work understandable to a wide audience, Section 2 explains architecture and microarchitecture, Section 3 side channels, Section 4 speculative execution, Section 5 Spectre and Section 6 its relevance. Section 7 argues why we should not seek the solution to the problem in software mitigations. One possible hardware fix for Spectre is to eliminate speculative execution, but the performance impact is unacceptably big (Section 8). Instead, a better fix is to eliminate the side channels back from the speculative world into the committed world (Section 9). Section 10 discusses the costs of this fix. Finally, Section 11 is a call to action for computer buyers, researchers and CPU manufacturers.

2 What is architecture and microarchitecture?

The architecture (aka instruction-set architecture, ISA) is the interface between the hardware and the software. Software sees main memory and registers, and instructions that work on them (see Fig. 1).

On the hardware side of this interface the highest design level is called microarchitecture. Microarchitecture is generally not visible in the functionality presented to the software, only through the performance. I.e., instructions generally deal only with architectural features such as memory and registers, not with microarchitectural features such as caches.[3]

E.g., the cache is a microarchitectural feature, and the CPU works functionally in the same way with the cache as without it (or with caches with different sizes); the only difference is that CPUs with caches run faster. While an access to main memory takes several hundred cycles on a modern general-purpose CPU, accessing the level-1 (L1) data (D) cache costs 3–5 cycles. However, the (L1) D-cache is much smaller (32-128KB on recent CPU cores), and contains only recently-accessed data.

[*]`anton@mips.complang.tuwien.ac.at`

[1]`https://www.anandtech.com/show/13450/intels-new-core-and-xeon-w-processors-fixes-for-spectre-meltdown`

[2]`https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html`

[3]There are a few cases where microarchitectural features are managed by software, and there are instructions for that, e.g., instructions for fetching data into caches (prefetch), instruction-cache management, or for draining the pipeline to ensure strictly in-order execution, but these instructions are not relevant for the rest of this paper.

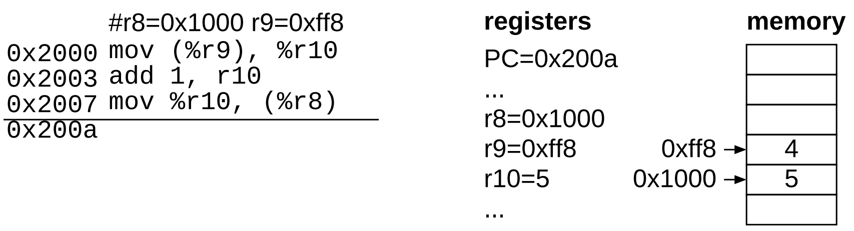


Figure 1: Architectural state: register and memory contents; this example shows the architectural state right after the instrucion at `0x2007`

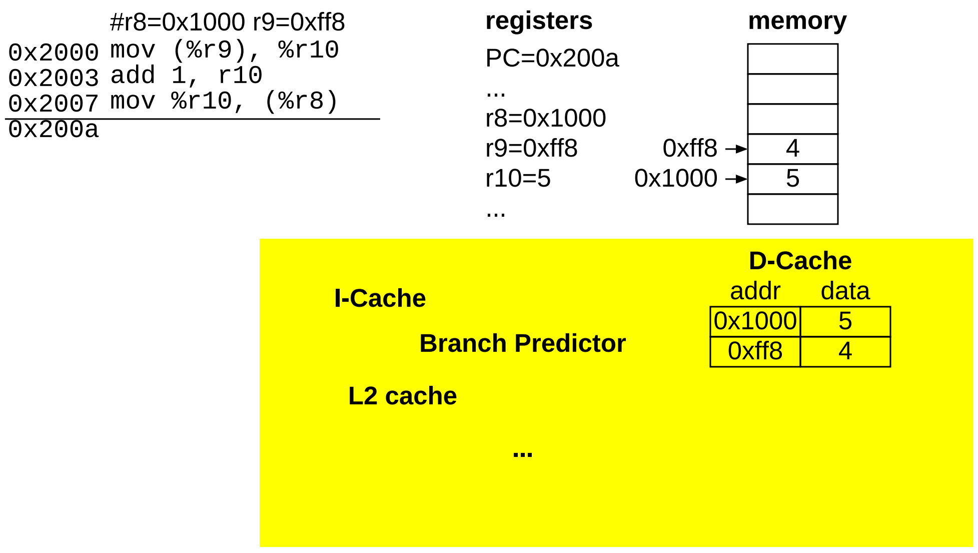


Figure 2: Microarchitecture (yellow background) is normally invisible to software, apart from its performance effects

Speculative execution is another microarchitectural feature and is discussed in Section 4.

3 What are side channels?

A side channel (aka covert channel) reveals data not directly by letting attacker read the secret data, but through ancillary properties of data processing.

E.g., if the run-time a program takes depends on the secret, an attacker can often use this fact to extract the secret (this kind of attack is known as a timing attack). E.g., a program could contain an `if`-statement where the condition depends on the secret, and the run-time of the two branches differs. For program code that deals with the dearest secrets (encryption keys and passwords), avoiding secret-dependent branches has long been best practice.

More generally, the best practice has been to write code that runs in constant time with respect to the secret.

The timing of memory accesses depends on the input address, thanks to caches. Caches provide such a big performance boost that we prefer to keep them and deal with the security implications in some other way rather than use CPUs without caches.

One case where memory access timing has played a role is AES encryption. It has been designed in a way that is hard to implement without loads from an address that depends on the secret key. While that dependence is quite convoluted, Bernstein has found a way to use the timing variation due to loads in such AES implementations to extract the key [Ber05].

3.1 Defending against side-channel attacks

The defense against side-channel attacks first requires realizing that there is a side-channel, and then taking measures that eliminate the leakage of secret information through that side channel.

As mentioned above, for timing side channels this has usually been done by writing the pieces of code that deal with the dearest secrets as constant-time code. These pieces of code tend to be miniscule (hundreds of lines) compared to the huge amounts of code (millions of lines) for complete programs like a web browser or an operating system.

While this makes the defense sound like being the responsibility of the software people alone (and this perception may have contributed to the lack of efforts on hardware fixes for Spectre), the software people cannot do it without support from hardware manufacturers.

In order to write constant-time code, the programmer needs guarantees that the timing of the used instructions does not depend on the input. Such guarantees have been historically hard to come by (and were only specified for specific implementations rather than the architecture), but recently Intel has guaranteed the input-independence of a subset of instructions for all of its implementations.[4]

In the AES case, the hardware manufacturers helped, not by making load timing address-independent (which would be impractical, as mentioned above), but by providing instructions that perform the problematic steps of AES in constant time without needing loads.

The discipline of writing constant-time code is used only for cryptographic and password-handling code, because it requires additional effort and specialized competencies, because it often results in slower run-time, but also because it is too limiting and impractical for implementing the requirements of most code. E.g., while the contents of spreadsheets of big companies and intelligence agencies may be considered by their users to be at least as secret as the encryption keys of ordinary citizens, to my knowledge nobody has tried to write a spreadsheet program with content-independent timing.

4 What is speculative execution?

Most modern general-purpose CPU core use speculative execution, a microarchitectural technique that works as follows:

The core's branch predictor predicts a likely future execution path and then executes (but does not commit) instructions on that path. The catch is that the prediction may turn out to be incorrect. In that case the architectural state (registers and memory) must not be changed in the way indicated by the misprediction prediction. If the speculation turns out to be correct, the changes can be committed (see Fig. 3).

Modern CPUs with speculative execution do this by conceptually dividing the core into a speculative part, which contains architectural results-to-be of unconfirmed speculative execution, and a committed part which contains the actual architectural state at the current architectural program counter (PC). So when the core architecturally processes an instruction (by committing (aka retiring) it in the reorder buffer), that instruction has often been speculatively executed some time earlier, and its result is lying around, waiting to be committed; and the commit takes this result and turns it into committed architectural state.

However, when a branch turns out to be mispredicted, and this branch is committed, all the speculative results after the branch (i.e., on the wrong path) are thrown away, and the processor starts executing on the correct path.

Note that this speculative execution not only contains register updates, but also stores to memory, possibly including several stores to the same memory location, and (speculative) loads from that location in between.

There have been many speculative-execution implementations of various architectures since the 1990s, and almost[5] all of them implemented the handling of architectural state correctly, both for correctly predicted branches and for mispredictions, for various kinds of registers, and for memory.

5 What is Spectre?

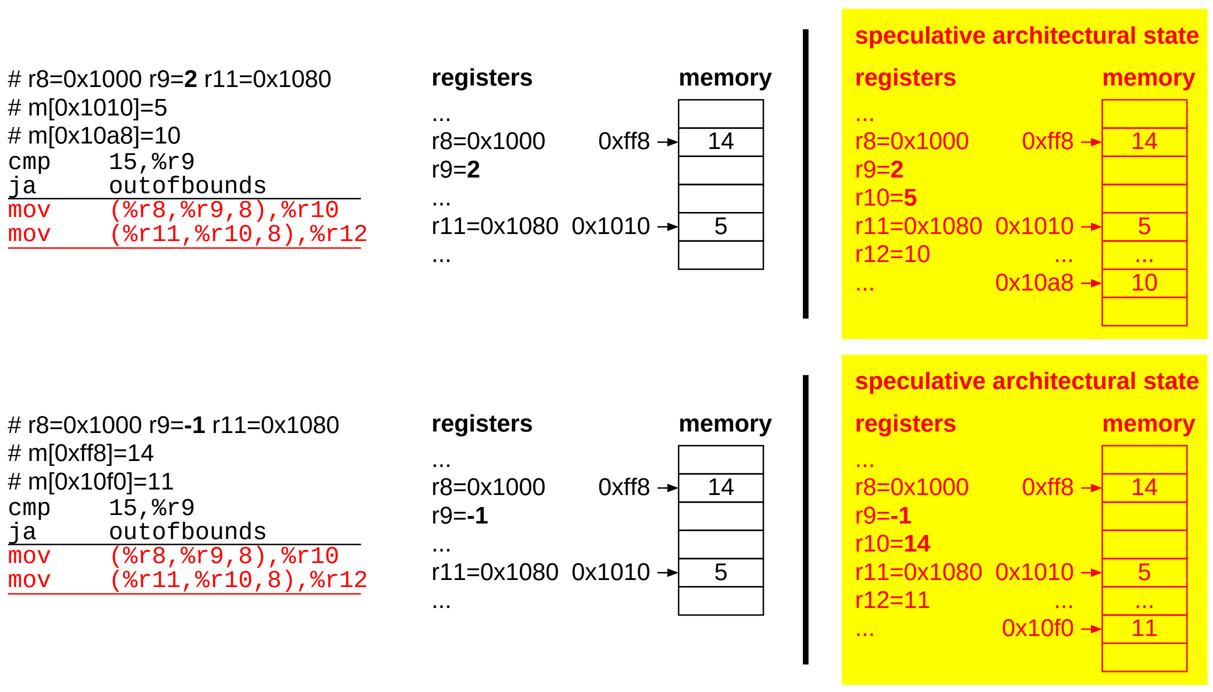
For microarchitectural state, e.g., the contents of the cache, existing processor cores do not discern between speculative and committed changes. After all, the idea is that microarchitectural state is invisible anyway. If a speculative load puts a line in the D-cache (and evicts another line), this has no architectural significance, so the hardware designers have had no qualms at performing this change speculatively, without a mechanism that cancels it in case of a misprediction.

Unfortunately, this approach opens a side channel that allows to leak data from the otherwise ephemeral world of misspeculation.

Figure 4 shows an example: The `cmp` and `ja` instructions architecturally prevent an out-of-bounds access to the array in `r8`, but if the branch is mispredicted to be not-taken, the following code is speculatively executed, and it reads the address `0xff8` speculatively; by using any other index, any other 64-bit value in the address space of the process could be accessed, including, e.g., secret keys or passwords that are there for cryptographic or authentication purposes. Let us assume that the secret is in memory location `0xff8`. In itself the load of the secret value into the speculative r10

[4] `https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html`

[5] The recently-published Zenbleed bug in AMD's Zen2 core (`https://lock.cmpxchg8b.com/zenbleed.html`) is the exception that proves the rule.



```
# r8=0x1000 r9=2 r11=0x1080
# m[0x1010]=5
# m[0x10a8]=10
cmp     15,%r9
ja      outofbounds
mov     (%r8,%r9,8),%r10
mov     (%r11,%r10,8),%r12
```

```
# r8=0x1000 r9=-1 r11=0x1080
# m[0xff8]=14
# m[0x10f0]=11
cmp     15,%r9
ja      outofbounds
mov     (%r8,%r9,8),%r10
mov     (%r11,%r10,8),%r12
```

Figure 3: Two examples of speculative execution, in both cases the `ja` instruction is predicted as being not taken. Above: The prediction is correct, and the speculative architectural state is eventually committed. Below: The prediction is incorrect, and the speculative architectural state is squashed.

does not appear a problem, because this is still the ephemeral world of misspeculation, and it cannot get out, right?

Unfortunately, on current hardware it can get out through a cache-based side channel: The second `mov` instruction loads a value into the D-cache, and the address of this load depends on the secret. The loaded cache line replaces a line that used to be in the cache, and which cache line is replaced depends on the address. An attacker can repeatedly access a number of memory locations in order to prime the cache, and can see from the timing of the cache accesses whether a cache line has been replaced, and in this way learn something about the secret.

There are a number of steps involved in Spectre attacks:

S1 The speculation itself: In this example (which is a Spectre v1 attack) a conditional branch causes speculative execution, but there are others. E.g., Spectre V2 uses indirect branches. There are also other speculative mechanisms in modern cores, such as speculating whether a memory load is to a different or the same address as an earlier store, and this has also been used in a number of attacks.

S2 The mechanism for getting the secret data into speculative architectural state. In the example above it is the load from `a[i]`. In the recently-published Downfall vulnerability [Mog23], it's gather instructions as implemented on some Intel microarchitectures.

S3 The sending side of the side channel for getting the data out of the misspeculation realm. In the example above it's the access to `b[j]` that channels information about `j` through the cache side channel. But other microarchitectural state can also be used, such as the power state of the AVX unit [SSLG18].

S4 The receiving side of that side channel. For a cache side-channel this consists of the attacker priming the cache and monitoring through timing which lines are replaced by the victim.

There is a lot of variation on all of these steps, leading to the stream of vulnerabilities that have been found up to now and continue to be found. For more details (and more aspects) there is a survey of the Spectre and Meltdown attacks until December 2020 [XS21]. A term that has been used to cover all these vulnerabilities and attacks against them is "transient execution vulnerabilities/attacks", but in this paper I just use "Spectre" in the same meaning as referring to all of these speculation-based side-channel attacks.

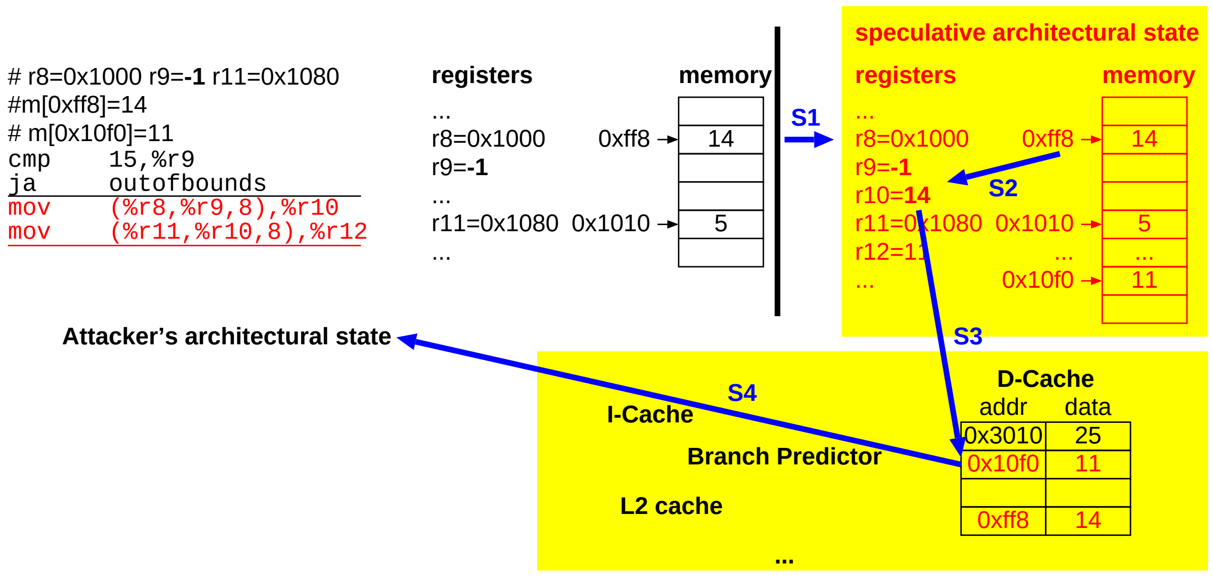


Figure 4: A Spectre v1 attack starts with a misprediction (S1), loads the secret (in `0xff8`) into speculative architectural state (S2), changes the cache state in a secret-dependent way (S3), and finally uses cache timing to extract the secret into the architectural state of the attacker (S4).

6 How relevant is Spectre?

Has Spectre been used in the wild? It's hard to know. Consider the case where attackers use Spectre to determine your password or encryption key. If they use that to decrypt your files, you may never know; maybe it was bad luck that your competitor undercut you by a narrow margin. But even if somebody does something very blatant like publish your documents on the Internet or encrypt your files and demand ransom, you usually don't know how the attacker got at your password or your encryption key.

However, a working exploit for reading normally unaccessible files on Linux has been discovered by Julien Voisin.[6] There is no proof that this exploit has been used for an actual attack, but given that it is widely available, it would be surprising if it has not.

Some people argue that Spectre is not relevant because there are many software vulnerabilities that may be used for subverting your system; so why, they argue, should an attacker bother with Spectre, which is supposedly harder to use. On the other hand, software vulnerabilities may be discovered and fixed at any moment, while Spectre exists unfixed on all desktop and server hardware, and is not even mitigated against in most software. So Spectre may be more attractive to attackers than some people give it credit for.

[6] https://dustri.org/b/spectre-exploits-in-the-wild.html

7 Why is software mitigation not a good way to deal with Spectre?

The mitigation of non-speculative timing attacks is to write the few hundred lines of code that deals with keys and passwords in a constant-time way. Can we not just deal with Spectre in the same way?

Unfortunately, for Spectre *all* the software that has the secret in its address space can potentially be used for an attack, and consequently would have to be hardened. For a web browser or an OS kernel that is typically millions of lines of code.

As an example of a mitigation, for the Spectre V1 example in Fig. 4, speculative load hardening has been proposed. A simple variant would change the code as follows:

```
     cmp    15,%r9
     ja     end
     mov    $0x0,%rax
     cmova  %rax, %r9
     mov    (%r8,%r9,8),%r10
     mov    (%r11,%r10,8),%r12
end:
```

Here the `cmova` hardens the following load, by setting `r9` to 0 if `r9>15`. While this condition cannot architecturally be true at that place, it can be true during misspeculation. The `cmova` instruction uses the same flags as the `ja` branch, but the mitigation assumes that `cmova` does not speculate.

In reality speculative load hardening is substantially more complicated, because it also has to also deal with possible speculation on earlier branches [ZBC+23].

Software mitigations have apparently led to the impression that Spectre is under control and no hardware fix is necessary, but they have a number of problems:

7.1 Still vulnerable

It has often turned out that many mitigations do not even completely close the vulnerability for which they are designed.

One reason for that is that the mitigation defends against a too-narrow attack scenario. E.g., speculative load hardening (SLH) has been implemented in the LLVM compiler and is intended to close the Spectre V1 vulnerability presented above, but it still has some leakage; this was then improved in Strong SLH [GP19] and recently Ultimate SLH [ZBC+23].

Another reason is that the mitigation relies on assumptions about microarchitectural mechanisms that turn out to be wrong; e.g., earlier Spectre V2 mitigations assumed that returns would only be predicted from the return stack, but there are some microarchitectures that use the general indirect-branch predictor to predict returns when the return stack is empty (so returns can also be used in Spectre V2 attacks).

Also, these mitigations tend to work only against a specific variant, but new variants are discovered all the time.

7.2 Performance

These mitigations cost performance, for the whole program (because with Spectre the whole program can be used to reveal the secret). E.g., Zhang et al. report a factor of around 2.5 slowdown (compared to no mitigation) for SPEC CPU 2017 (int and fp, rate and speed) [ZBC+23]. I saw a slowdown (compared to using no mitigation) by a factor 2–9.5 from compiling Gforth with the fastest retpoline mitigation against Spectre V2[7].

7.3 Effort

Because the slowdowns that you get from applying compiler-based mitigations across the board are often considered to be unacceptable, there is the idea that programmers should be more selective and analyse whether each specific place in a program can actually be used by an attacker, and only harden those places, lowering the performance cost.

However, this requires a huge amount of effort, and it takes only one place in the potential attack surface that the programmer mistakenly has not hardened, and the program is still vulnerable.

And when the next vulnerability and mitigation shows up, you have to do it all again. And when the program is changed (due to, e.g., new requirements), you have to analyse more than just the changed lines.[8]

8 Why is the first idea for a fix not so great?

The first idea many people have for fixing Spectre is to eliminate speculative execution. While this certainly fixes Spectre by preventing step 1 of the exploits, the performance impact of this measure is pretty bad: E.g., the core without speculation that shows the best performance on our LaTeXbenchmark[9] is the 1800MHz Cortex-A55 on the Rock 5B single-board computer. The Cortex-A76 core (with speculative execution) running at 2275MHz on the same computer is 3.3× as fast for this benchmark, and the 3000MHz Apple Firestorm is 7.8× as fast.

Given the performance impact, it's no surprise that we have not seen a resurgence of microarchitectures without speculation. The number of customers that would exchange this much performance for security against Spectre is small. The customers' reasoning is as follows: There are lots of vulnerabilities in the software we use, so fixing Spectre is not going to make our computers that much safer. Therefore we are not willing to sacrifice that much performance for this benefit.

9 How to fix Spectre in hardware?

The less costly and therefore better way to fix Spectre is to prevent step S3.

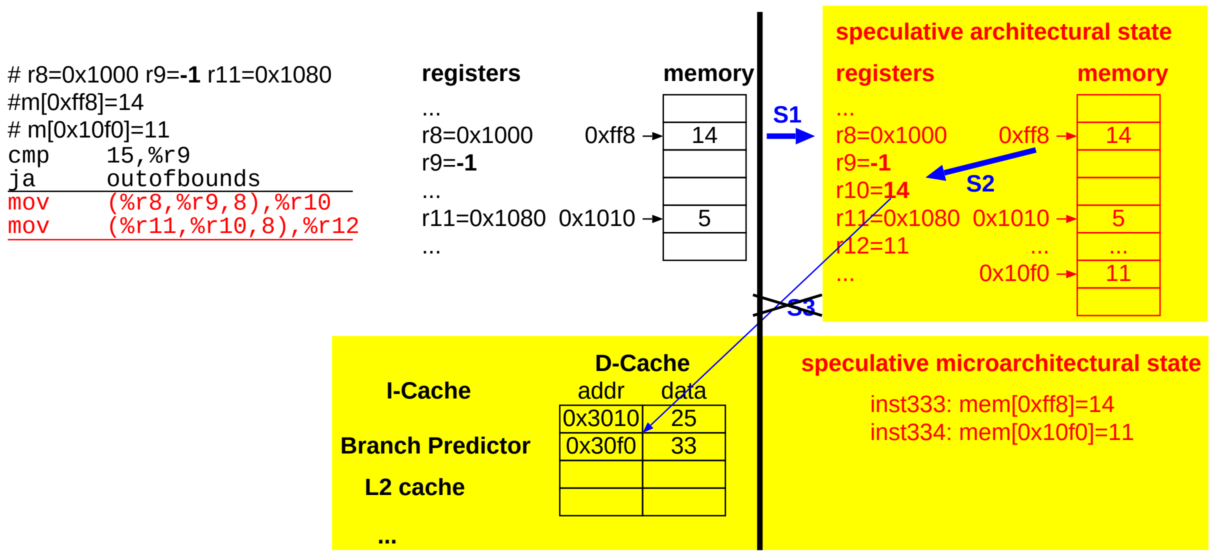
9.1 Side channels based on microarchitectural state

In particular, for the side channels through microarchitectural state, we can use the same approach for microarchitectural state as for architectural state: keep the speculative state separate from the committed state, and squash it when it turns out that the speculation is wrong. This goes for all microarchitectural state: D-cache, I-cache, branch predictor, TLBs, etc.

[7] news://<2023Jan15.105348@mips.complang.tuwien.ac.at>

[8] The idea that you do not have to reanalyse code when the requirements change resulted in the Ariane 501 explosion.

[9] https://www.complang.tuwien.ac.at/franz/latex-bench



```
# r8=0x1000 r9=-1 r11=0x1080
#m[0xff8]=14
# m[0x10f0]=11
cmp     15,%r9
ja      outofbounds
mov     (%r8,%r9,8),%r10
mov     (%r11,%r10,8),%r12
```

Figure 5: Separating speculative microarchitectural state from committed microarchitectural state eliminates the S3 part (and therefore also S4) of a Spectre attack, as far as microarchitectural state is concerned; resource-limitation side channels also need to be addressed, see text.

In the case of D-cache, several papers [YCS+18, KKS+19, AJ20] have already proposed ways of doing that, probably because the cache side channel has been the most popular one for Spectre-type attacks. But of course, the other state-based side channels need to be closed, too.

Some attempts at fixes for state-based side channels have tried to undo the changes when a speculation turns out to be false, e.g., CleanupSpec [SQ19]. However, I think that it is better to keep the speculative changes separate until commit time, for the following reasons:

- The microarchitectural state is changed, albeit only for a short time, and this is visible to potential attackers, given enough effort.

- It is harder to reason about the correctness of an undoing approach than about an approach that never speculatively changes the non-speculative state.

- Undoing approaches have been tried for architectural state [DA92], but committing approaches have won. It is likely that the same reasons will make undoing of microarchitectural state unattractive.

9.2 Side channels based on resource contention

Apart from the popular state-based side channels, another kind of side channel is contention for resources such as execution ports, functional units, or cache ports. SMoTherSpectre [BSN+19] attacks another SMT thread on the same core by using execution port contention by the speculatively executing victim as a side channel. Even worse, speculative interference attacks [BSP+21] use resource contention to affect the timing of an older (eventually committing) instruction in the same thread.

For the SMT side channel, a solution is to have a fixed partitioning of resources between the threads, so that no thread can use resource contention as a side channel. This means that resources that exist only once have to be time-shared. E.g., if there is an non-pipelined divider that takes 20 cycles for the division, there are fixed 20-cycle time slots for each thread, and when a thread does not have a division ready at the start of its time slot, that time slot goes unused. This fixed partitioning will cost some performance; it could be made optional, allowing the full benefit of SMT to be used in settings where the sibling threads are believed to not spy on each other.

For the same-thread problem, Behnia et al. [BSP+21] describe the high-level principle: “a speculative instruction must not influence the execution of a non-speculative instruction”. And they describe two rules that ensure that:

- “No instruction ever influences the execution time of an older instruction.” They propose to achieve this by giving priority to older instructions in case of resource contention. They discuss several options how to deal with non-pipelined execution units. The slot idea above is another way to deal with that: If a thread can start using the execution unit only at the start of a slot, the priority approach works for non-pipelined units (although one might wish for better performance).

- "Any resources allocated to an instruction at the front end and the execution engine are not deallocated until the instruction becomes non-speculative". This rule ensures that misspeculated code cannot produce timing variations by congesting the front end.

9.3 Other side channels

Another known side channel is energy consumption. In particular, Meltdown-Power [KJG+23] uses speculation for S1 and S2, and then a power-based side channel for S3 and S4. However, it requires that the speculative load updates the cache, which does not happen with the fix for speculative microarchitectural state outlined above, so fixed hardware would be immune against this particular attack.

Still, one can imagine that the energy consumption of e.g., functional units working on misspeculatively loaded data could reveal something about the data. At the moment I have no good hardware answer for that. On the other hand, the question is if such an attacks can be made practical (i.e., leak relevant amounts of data in realistic time frames).

10 How much does the fix cost?

The fixes certainly cost design complexity. Hardware architects have been remarkably good at handling the increasing complexity of modern high-performance CPUs, and I expect them to rise to the challenge of designing fixed hardware, if they are given the task.

The resulting CPU cores will require more area, for the speculative state. E.g., if we want to be able to buffer, say, 30 cache lines loaded from outer cache levels in speculative microarchitectural state, the memory for these 30 cache lines is needed, as well as the infrastructure to look up data in them and deal with snoop messages. Compared to the 224 physical ZMM registers (each with 64 bytes) in Intel's Sunny Cove core, this does not seem to add that much area; and I expect that the area for other microarchitectural features will be even smaller.

Concerning performance, the additional buffers can even help, and for MuonTrap [AJ20] the Parsec benchmarks indeed see a speedup by a factor 1.05. However, SPEC 2006 sees a slowdown by a factor 1.04 compared to vulnerable hardware. And then there is the question of how much speed the additional area could have produced if it was invested just in performance. On the other hand, compared to applying software mitigations to all software (e.g., a factor 2.5 for defending only against Spectre v1), even the SPEC slowdown and the opportunity performance cost of the additional area are small.

One may want to compare with the more selective hardening approach that is used in, e.g., the Linux kernel. This kind of hardening has not been applied to the SPEC benchmarks, and the hardware fixes have not been measured on the benchmarks that are typically used for measuring the Linux kernel performance, so a direct comparison is not possible. Looking at Michael Larabel's results for how the kernel mitigation of just Inception[10] and the firmware mitigation of just Downfall [11] slows down applications, the slowdowns are often larger than what has been reported as slowdown from hardware fixes for the cache side channel. While these are numbers for different programs and mitigations/fixes for different vulnerabilities, and both comprehensive software mitigations and comprehensive hardware fixes will have higher cost, I expect that the majority of the performance cost of a hardware fix is in dealing with the cache (because of stuff like cache coherence), so I don't expect the cost of a comprehensive hardware fix to be that much higher than the cache-only approaches we have seen yet, while on the software mitigation side, every vulnerability seems to require its own mitigation, with a program-dependent performance impact, sometimes very expensive, as discussed above.

11 What should I do?

As computer **customers**, we should keep asking the CPU manufacturers when they will finally fix Spectre in hardware; we should tell them that software mitigations are not good enough.

And when one of the manufacturers comes out with a CPU with a Spectre fix, we should prefer these CPUs in our buying decisions even if they are a little slower at running unmitigated software (or software with mitigations that are unnecessary for the fixed CPUs). After all, such a CPU will be safer than an unfixed CPU when both run unmitigated software (the usual case). And such a CPU will be faster and at least as safe (probably safer) when the fixed CPU runs software without mitigations and the unfixed hardware runs software with mitigations.

When CPU manufacturers claim that they have fixed Spectre, only believe them when they explain how they did it (and only if that explanation does not have holes); don't accept hand-waving along

[10] `https://www.phoronix.com/review/amd-inception-benchmarks`
[11] `https://www.phoronix.com/review/intel-downfall-benchmarks`

the lines of "Differences in AMD architecture mean there is a near zero risk of exploitation"[12].

As **computer architecture researcher**, you can work at designing and evaluating mechanisms for fixing Spectre. Even if there is already some work in that direction, there is probably still some microarchitectural state or other side channels that have not been covered yet. And even for the microarchitectural state that has been covered, there are probably ways to improve on it, i.e., a solution that costs less area and/or less performance.

If your research leans more towards **theory**, you could work out a formal description of speculative side channels, and a way how computer architects could prove that they have closed these side channels. I do not know if they worked out such an approach to make sure that speculation works correctly for architectural state; it may be (usually) good enough to validate the architectural design by running test programs, but for microarchitectural state and other side channels, such an approach is needed, because the side channel does not show up in the usual architectural validation.

If you work at a **CPU manufacturer** (or CPU design house), you have the best opportunity to fix this problem. If the decision is up to you, go ahead and decide that you will make a Spectre-immune high-performance CPU core. If the decision is up to someone else, make a case that convinces them that the fix is worth the development and manufacturing costs by making your CPU safer than the competition, and to put a stop to the constant stream of new Spectre- and Meltdown-type vulnerabilities (and the slowdowns from firmware and software mitigations). Also, imagine what happens if your competition is first at presenting a Spectre-immune CPU.

12 Conclusion

Attacks like Spectre that extract speculative state through a side channel are different from earlier side-channel attacks in being impractical to mitigate in software: not just the small piece of code that deals with the secret, but all software in the same address space as the secret (including libraries) needs to mitigate these attacks; E.g., an automatic compiler approach against Spectre v1 alone costs a factor 2.5 in performance, and that does not defend against all Spectre attacks (e.g., not against Spectre v2). One way to reduce this cost taken in, e.g., the Linux kernel, is to try to identify places that can be attacked and only harden those; this costs a lot of programmer effort, has the potential danger of leaving a hole open, and when another attack is discovered, this effort often has to be repeated.

Therefore the right way to deal with Spectre is to fix it in hardware. For speculative microarchitectural state, it should be treated just like speculative architectural state: During speculation, keep it separate from the committed state; and when the speculation turns out to be wrong, just squash the speculative state (including speculative microarchitectural state). When the speculation is correct, turn the speculative state into commited state (e.g., during instruction commit).

In addition to state-based side channels, resource contention can also provide a side channel. This can be addressed with a fixed partitioning of resources in an SMT setting, by always prioritizing older instructions in resource conflicts, and by managing front-end resources in a specific way.

A hardware fix for Spectre costs some chip area and often also performance compared to a vulnerable core, but much less than applying a software mitigation against just Spectre v1 across the board.

[12] `https://web.archive.org/web/20180104014617/https://www.amd.com/en/corporate/speculative-execution`

References

[AJ20] Sam Ainsworth and Timothy M. Jones. MuonTrap: Preventing cross-domain Spectre-like attacks by capturing speculative state. In *International Symposium on Computer Architecture (ISCA)*, pages 132–144, 2020. 9.1, 10

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES. 2005. 3

[BSN+19] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *Conference on Computer and Communications Security*, 2019. 9.2

[BSP+21] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christo pher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. Speculative interference attacks: Breaking invisible speculation schemes. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, pages 1046–1060, 2021. 9.2

[DA92] Keith Diefendorff and Michael Allen. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, pages 40–63, April 1992. 9.1

[GP19] Marco Guarnieri and Marco Patrignani. Exorcising Spectres with secure compilers. *CoRR*, abs/1910.08607, 2019. 7.1

[KJG+23] Andreas Kogler, Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Collide+Power: Leaking inaccessible data with software-based power side channels. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7285–7302, Anaheim, CA, August 2023. USENIX Association. 9.3

[KKS+19] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with leakage-free speculation. In *Design Automation Conference*, 2019. 9.1

[Mog23] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *32th USENIX Security Symposium (USENIX Security 2023)*, 2023. 1, 5

[SQ19] Gururaj Saileshwar and Moinuddin K. Qureshi. CleanupSpec: An undo approach to safe speculation. In *International Symposium on Microarchitecture*, page 73–86, 2019. 9.1

[SSLG18] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. *CoRR*, abs/1807.10535, 2018. 1, 5

[TWR23] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. Inception: Exposing new attack surfaces with training in transient execution. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7303–7320, Anaheim, CA, August 2023. USENIX Association. 1

[XS21] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. *ACM Computing Surveys*, 54(3), May 2021. 5

[YCS+18] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *International Symposium on Microarchitecture*, page 428–441, 2018. 9.1

[ZBC+23] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate SLH: Taking speculative load hardening to the next level. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7125–7142, Anaheim, CA, August 2023. USENIX Association. 7, 7.1, 7.2

4g and FAIL
(or: Be careful what you joke about!)

Glyn Faulkner
EuroForth 2023

2023-09-15

Historical context

My final slide last year:

Where next?
How about a parameterised Forth interpreter generator?

```
[marsu@celaeno 4g]$ ./4g -t ITC -T -m ANSI -o forth
Indirect-threaded x86_64 Linux ANSI Forth
Options: top-of-stack in register, linked-list dictionary
Generating forth.S
gcc -m64 forth.S -o forth
Done
[marsu@celaeno 4g]$ ./forth
```

Ask me how this is going next year!

This is how it's going. . .

4g, the Forth-generator

Used approach from Peter Knaggs' EuroForth paper[1] to build a "matched-pair" of ANSI-ish Forths, sharing as much source code as possible:

Direct-threaded:

```
.macro $next
    lodsl
    jmp *%eax
.endm
#include "common.S"
.section .flat
.align 4
docol:
    $pushrs %esi
    pop %esi
$next
```

Indirect-threaded:

```
.macro $next
    lodsl
    jmp *(%eax)
.endm
#include "common.S"
.section .text
.align 4
docol:
    $pushrs %esi
    lea 4(%eax), %esi
$next
```

[1]Peter Knaggs *Using Test Driven Development to build a new Forth interpreter*, http://www.euroforth.org/ef21/papers/knaggs.pdf

Problems...

- Not a scaleable approach!
  - new-runtime system required for every possible configuration or mapping of Forth to machine registers
  - multiple inter-dependent source files makes development slow and painful

It gets worse...

"common.S" is not so common! With SP mapped to %esp, dup looks like this...

```
code dup  ( x -- x x )
    pop %eax
    push %eax
    push %eax
end-code
```

...but with SP in %esi it might look like this...

```
code dup  ( x -- x x )
    lodsl
    lea -4(%esi), %esi
    mov %eax, (%esi)
    lea -4(%esi), %esi
    mov %eax, (%esi)
end-code
```

Assembler Macros!

```
code dup  ( x -- x x )
    $popds RegX
    $pushds RegY
    $pushds RegY
end-code
```

But if we have top-of-stack in a register, we want dup to look something like this:

```
code dup  ( x -- x x )
    push %ebx
end-code
```

What reasonable definition of $pushds and $popds can give us this?

- Macro complexity rapidly explodes!
- Debugging becomes a nightmare.
- Generated code is hard to read and modify.

What a mess!

F.A.I.L.: The Forth Abstract Instruction Language

The original insight:

```
code dup  ( x -- x x )
    ...
```

Hmm. That stack comment looks suspiciously compilable!

```
    ( x -- x x )

    1. Move top-of-stack to register X
    2. Re-pack the stack with two copies of register X
```

Stack shuffling 1

```
: dup  (   x -- x x ) ;
```

. . . becomes. . .

```
code dup
    # pop stack to x
    pop %eax
    # push x to stack twice
    push %eax
    push %eax
    $next
end-code
```

Stack shuffling 2

Data stack pointer is somewhere exotic? No problem?

```
code dup
    # pop stack to x
    mov (%eax), %edi
    add $4, %eax
    # push x to stack twice
    sub $4, %eax
    mov %edi, (%eax)
    sub $4, %eax
    mov %edi, (%eax)
end-code
```

Stack shuffling 2

Data stack pointer is somewhere exotic? No problem?

```
code dup
    # pop stack to x
    mov (%eax), %edi
    add $4, %eax
    # push x to stack twice
    sub $4, %eax
    mov %edi, (%eax)
    sub $4, %eax
    mov %edi, (%eax)
end-code
```

The assembly output is less than optimal, but quite readable.

Recall that my goal is to automate the boring parts of bringing up a new Forth-like language.

Return stack 1

Works also with the return stack

```
: >r   ( x --   ) (r:    -- x ) ;
```

```
code >r
    # pop stack to x
    pop %eax
    # push x to return stack
    sub $4, %ebp
    mov %eax, (%ebp)
    $next
end-code
```

Return stack 2

Using both stacks at once. . .

```
: r@ (   -- x ) (r: x -- x ) ;
```

```
code r@
    mov (%ebp), %eax   ;  add $4, %ebp
    push %eax
    sub $4, %ebp   ;  mov %eax, (%ebp)
    $next
end-code
```

Return stack 3

What about a word with atypical behaviour. . .

```
: exit    (r: IP --   ) ;
```

(IP is the hardware register holding the Forth instruction pointer)

```
code exit
    # pop return stack to IP
    mov (%ebp), %esi
    add $4, %ebp
    $next
end-code
```

Words that actually *do* something 1

Stack shuffling isn't Turning complete! (probably)

Words that actually *do* something 1

Stack shuffling isn't Turning complete! (probably)

We can already map the stack onto virtual registers. . .

```
: dup  (   x -- x x ) ;
```

. . . which map onto machine-registers.

```
: dup  (   %eax -- %eax %eax ) ;
```

So what if we imagine an assembly-like syntax that works on virtual registers?

```
: +    ( a b -- c )    a b  c  +   ;
```

Borrowing ideas from QEmu's TCG intermediate representation, (almost) all of my primitives have separate parameters for source and destination registers.

(I regret my syntax choice here: the first + is defining the Forth word and the second is a FAIL primitive).

Words that actually *do* something 2

The resulting assembly looks like this...

```
code +
    pop %ecx
    pop %eax
    add %ecx, %eax
    push %eax
    $next
end-code
```

push and pop

push and pop have their own instructions (of course!).

```
<dest> <ptr> pop
<src> <ptr> push
```

Special case:

- the *only* FAIL instructions that modify a register in-place.
- `<ptr>` follows target register in both cases for ease of reading which stack is being accessed.

Common uses:

```
\ pop the data-stack to x
x  SP pop
\ push y to the return-stack
y  RP push
\ threaded-code NEXT
W  IP pop
W     execute
```

Branching

```
: (brn) (   -- )
    IP IP @
;
: (brz) ( n -- )
    \ "pop" through IP
    \ to reg b
    b  IP pop
    n  (0=) if
        b IP move
    then
;
```

```
code (brn)
    mov    (%esi), %esi
    $next
end-code
code (brz)
    pop %eax
    mov (%esi), %ecx
    add $4, %esi
    test %eax, %eax
    jnz 1f
        mov     %ecx,  %esi
    1:
    $next
end-code
```

Condition specification syntax using (0=) as an “argument” to if is awkward. Is there a better way?

Complications! 1

x86 has some *nasty* instructions:

- `div` and `idiv` have four implicit arguments (two source and two destination)
- `mul` and `imul` can clobber `%edx` (which might be Forth's stack- or instruction-pointer!)
- `shl` `shr` `sar` and `sal` require the number of places shifted to be in `%cl`

. . . and lots of register aliasing:

- `%eax`, `%ax`, `%ah` and `%al` all refer to the same hardware register!

```
: sm/rem  ( a b c -- d e )
    a b c  d e  sm/rem ;
```

. . . needs to produce something like this:

```
code sm/rem  ( l h d -- r q )
    pop %ecx
    pop %edx
    pop %eax
    idiv %ecx
    push %edx
    push %eax
    $next
end-code
```

`h`, `l`, `r`, and `q` *must* be in the correct machine registers.

How to solve?

Complications! 2

Current answer: cheat!

- Check abstract instruction's register affinity
- Run rudimentary liveness analysis.
- Allocate required registers if possible.
- Otherwise throw a compilation error.

Two possible solutions

- QEmu-style "helper functions"
  - compilation guaranteed to succeed
  - **but** run-time overhead of switching out of "Forth-mode" and into e.g. "C-mode"
- More advanced register allocator
  - spilling registers could handle some tricky cases
  - some x86 instructions can work directly with a value in memory, no register allocation required.
  - **but** compilation failure is still a possibility
  - **unanswered questions:** how does spilling work if our stack pointers are the registers we want to spill?

What next?

F.A.I.L.

- Port the start-up code for the runtime to FAIL.
- Easier configuration (currently requires editing an Awk script!)
- Use FAIL words inside FAIL words (currently `$next` has to be an assembly macro!)
- Better register allocation
- Support more threading models: Token, Subroutine. . .
- x86_64 and ARM support
- Re-write in Forth! (currently Awk!)
- More flexible instruction generation (optimise for size, speed, readability. . .)
- Abstract away the dictionary implementation
- Selectable back-end (GNU as, nasm, C, machine code. . .)

4g

- Package as a commandline tool (currently a Makefile!)
- More complete and correct ANSI support
- Add other Forth "models": eForth, F83, F77?
- Port some of my own Forths!

Any Questions?

The Performance Effects of Virtual-Machine Instruction Pointer Updates

M. Anton Ertl, TU Wien

Example: `matrix.fs`

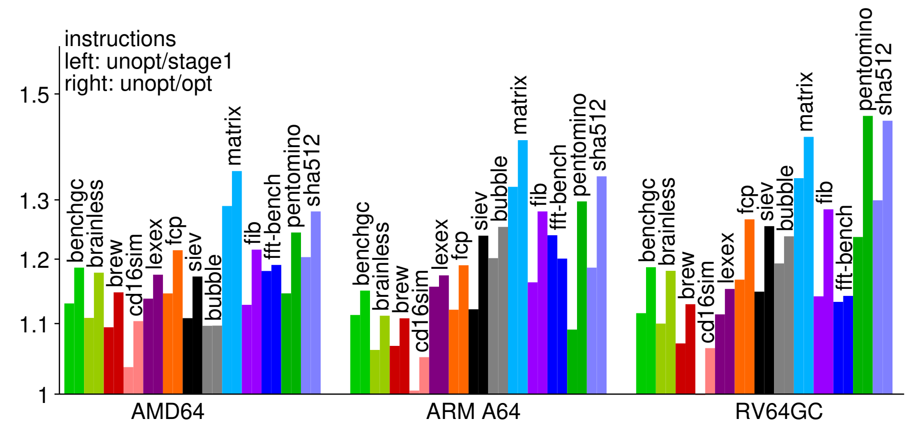
```
: innerproduct ( a[row][*] b[*][column] -- int)
  0 row-size 0 do
    >r over @ over @ * r> + >r
    swap cell+ swap row-byte-size +
    r>
  loop
  >r 2drop r>
;
```

threaded code		unopt.	opt.	threaded code		unopt.	opt.
>r	1->1	sd s7,-8(rp)	sd s7,-8(rp)	cell+	2->2	*addi ip,ip,8*	
		ld s7,8(sp)	ld s7,8(sp)			addi s0,s0,8	addi s0,s0,8
		addi sp,sp,8	addi sp,sp,8	swap	2->1	addi sp,sp,-8	addi sp,sp,-8
		addi rp,rp,-8	addi rp,rp,-8			*addi ip,ip,8*	
		addi ip,ip,8				sd s0,8(sp)	sd s0,8(sp)
over	1->2	ld s0,8(sp)	ld s0,8(sp)	lit+	1->1	ld a5,*0(ip)*	ld a5,*104(ip)*
		addi ip,ip,8		1600		*addi ip,ip,16*	
@	2->2	ld s0,0(s0)	ld s0,0(s0)			add s7,s7,a5	add s7,s7,a5
		addi ip,ip,8		r>	1->1	sd s7,0(sp)	sd s7,0(sp)
over	2->3	mv s3,s7	mv s3,s7			addi sp,sp,-8	addi sp,sp,-8
		addi ip,ip,8				ld s7,0(rp)	ld s7,0(rp)
@	3->3	ld s3,0(s3)	ld s3,0(s3)			*addi ip,ip,8*	
		addi ip,ip,8				addi rp,rp,8	addi rp,rp,8
*	3->2	mul s0,s0,s3	mul s0,s0,s3	(loop)	1->1	ld a5,0(rp)	ld a5,0(rp)
		addi ip,ip,8		start		ld a4,8(rp)	ld a4,8(rp)
r>	2->3	ld s3,0(rp)	ld s3,0(rp)			ld a3,*0(ip)*	ld a3,*128(ip)*
		addi ip,ip,8				addi a5,a5,1	addi a5,a5,1
		addi rp,rp,8	addi rp,rp,8			*addi a6,ip,8*	sd a5,0(rp)
+	3->2	add s0,s0,s3	add s0,s0,s3			beq a4,a5,end	beq a4,a5,end
		addi ip,ip,8				sd a5,0(rp)	
>r	2->1	addi rp,rp,-8	addi rp,rp,-8			ld a4,0(a3)	ld a4,0(a3)
		addi ip,ip,8				*addi ip,a3,8*	*mv ip,a3*
		sd s0,0(rp)	sd s0,0(rp)			jr a4	jr a4
swap	1->2	ld s0,8(sp)	ld s0,8(sp)			end:	end:
		addi ip,ip,8				*addi ip,a6,8*	
		addi sp,sp,8	addi sp,sp,8			sd a5,0(rp)	

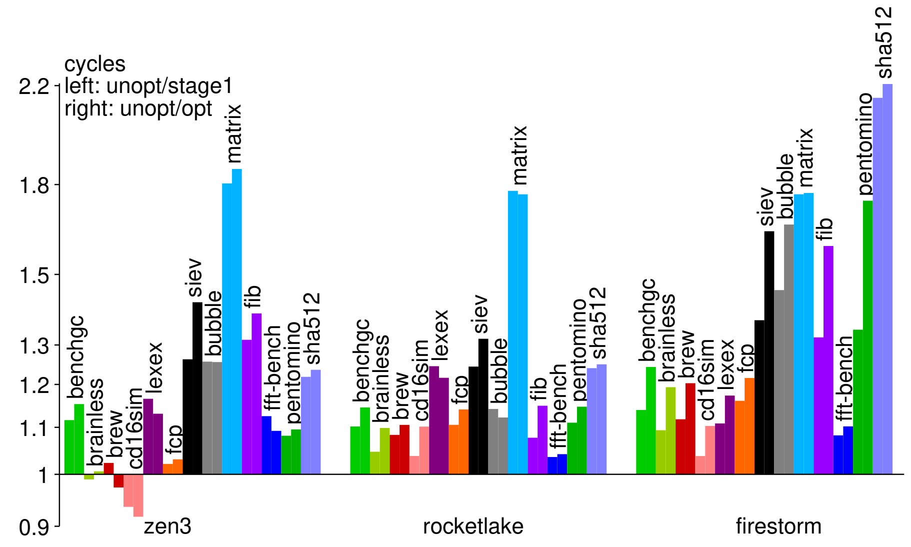
Optimize away most `ip` updates

- Normal case: Don't insert ip-update
- Remember which threaded-code cell `ip` points to
- If `ip` must be up-to-date, insert ip-update
  (Taken branch)
  Superblock end
  Calls
  non-relocatable native code
  immediate arguments (in some cases)
- Versions of ip-updates for 1–24 cells
- Versions of primitives with immediate arguments with varying `ip` offsets
  `lit call ?branch lit@ branch (loop) lit-perform lit+ does-xt`
- No architecture-specific code

Instructions



Performance (modern high-performance cores)



Questions

- Why is the speedup much higher than instruction reduction (for some benchmarks)?
- Why only for some benchmarks?
- Why is the speedup of sha512 bigger on Firestorm?

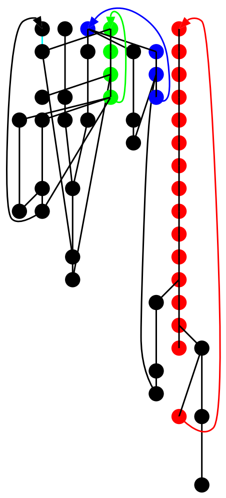
6

Performance components

- Branch mispredictions (solved)
- Resource limitations
- VM data dependences
- sp-update dependences
- rp-update dependences
- *ip-update* dependences

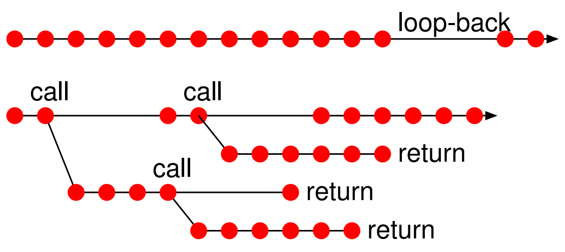
7

Data flow graph of `innerproduct`

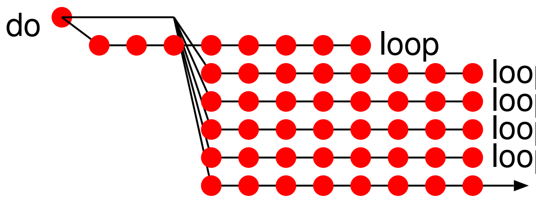


8

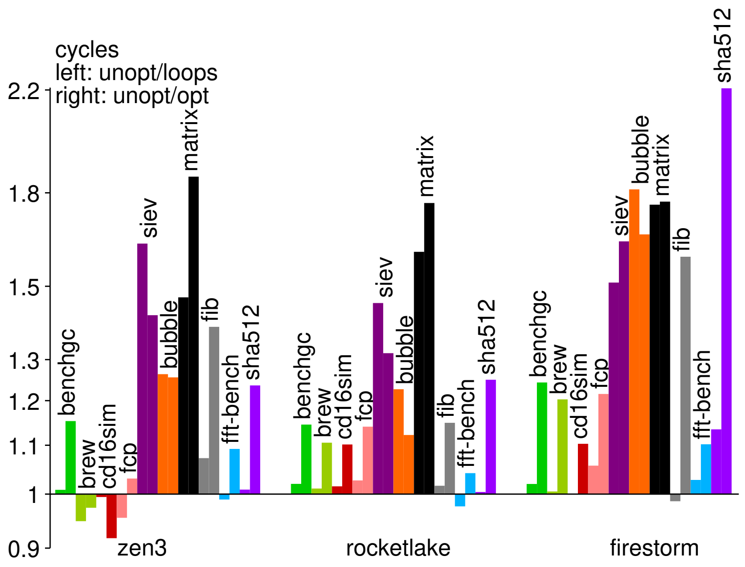
Why do we not see such speedups for all benchmarks?



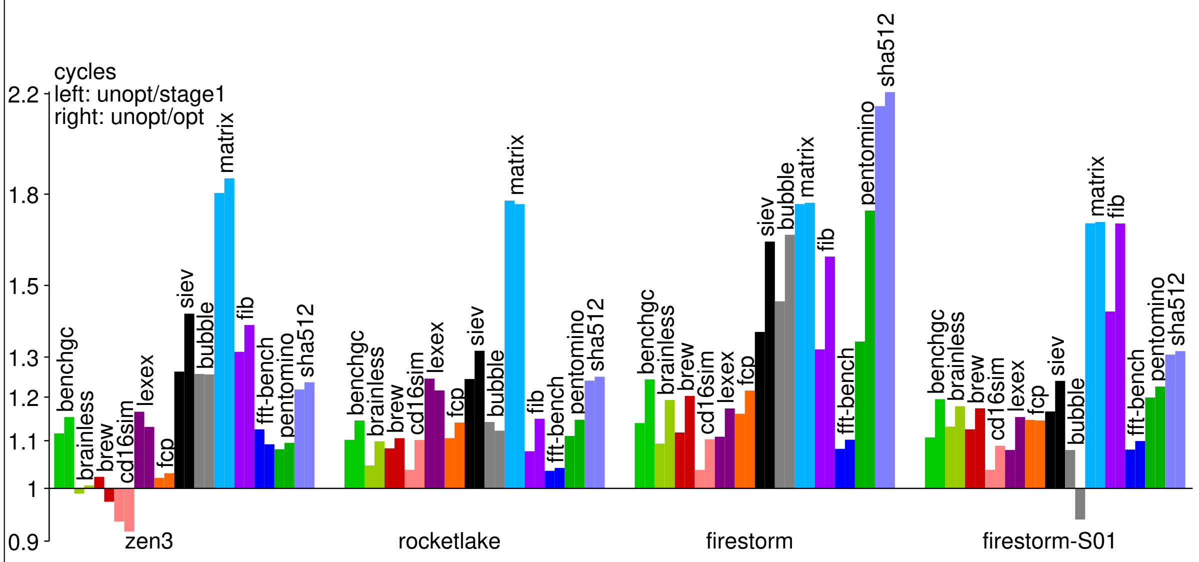
Alternative: Keep `do` loop-back address on return stack



Performance of `do` loop alternative vs. ip-update optimization



Why is the speedup of sha512 bigger on Firestorm? Stack caching!



12

Conclusion

- Optimizing ip updates can be done portably
- Reduces executed instructions by ≈ 1.2×
- Increases performance by up to 2.2×
  because ip-updates are the critical path in looping benchmarks
  Alternative: optimize loops
- Synergy between stack caching and ip-update optimization

13