21st EuroForth Conference

October 21-23, 2005

Santander, Spain

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 21th EuroForth finds us in Santander for the first time. The two previous EuroForths were held in Ross-on-Wye (2003) and in Schloss Dagstuhl (2004). Information on earlier conferences can be found at the EuroForth home page (http://dec.bournemouth.ac.uk/forth/euro/index.html).

Since 1994, EuroForth has a refereed and a non-refereed track. This year, no papers were submitted to the program chair for refereeing.

A number of papers were submitted to the non-refereed track in time to be included in these proceedings. In addition, a number of abstracts were submitted for talks at the conference. Workshops and social events complement the program. The conference is preceded by a Forth200x standards meeting.

We are grateful to Federico de Ceballos for organizing this year's EuroForth.

Anton Ertl

Program committee

Sergey N. Baranov, Motorola ZAO, Russia
M. Anton Ertl, TU Wien (chair)
David Gregg, University of Dublin, Trinity College
Phil Koopman, Carnegie Mellon University
Jaanus Pöial, Estonian Information Technology College, Tallinn
Bradford Rodriguez, T-Recursive Technology
Reuben Thomas

Contents

Abstracts

Type checking FORTH

Jürgen Pfitzenmaier

Part I

The ANS standard for FORTH states in A.3.1.3.3 the one-to-one relationship between addresses and unsigned numbers. This relationship can be applied in two different ways: Using this relationship a lot (this is backed by the reading of the current standard) or making only sparse use of this relationship. We show that this relationship can make type checking FORTH unsound. A few conservative changes to the standard would suppress the relationship in most cases and enable a sound type checking without the need to change an existing FORTH implementation. We give examples showing the problems in the current standard and the proposed solutions.

Part II

The stack notations in the current ANS standard for FORTH are neither sufficient to describe the role of execution tokens nor sufficient to describe the semantic actions of deferred execution, or of the words IF and DOES¿ when it comes to type checking. A detailed example shows the necessary conservative changes to the standard and how IF can be type checked.

Part III

Full type checking of a FORTH program needs (at least) one nonconservative deviation from the ANS standard: the implementation dependent size on stack of the data types colon-sys, do-sys, ..., nest-sys must be looked up by an environmental query.

Stack Caching in Forth

M. Anton Ertl*
TU Wien

David Gregg
University of Dublin, Trinity College

Abstract

Stack caching speeds Forth up by keeping stack items in registers, reducing the number of memory accesses for stack items. This paper describes our work on extending Gforth's stack caching implementation to support more than one register in the canonical state, and presents timing results for the resulting Forth system. For single-representation stack caches, keeping just one stack item in registers is usually best, and provides speedups up to a factor of 2.84 over the straight-forward stack representation. For stack caches with multiple stack representations, using the one-register representation as canonical representation is usually optimal, resulting in an overall speedup of up to a factor of 3.80 (and up to a factor of 1.53 over single-representation stack caching).

1 Introduction

In threaded-code interpreters for Forth, and especially in simple inline-expanding native-code compilers a significant part of the run-time is consumed by loading stack items from and storing them to memory, and by stack pointer updates.

A frequent technique for reducing that overhead is to keep the top-of-stack in a register. Stack caching [Ert95] is a generalization of this technique. In the past we have presented data based on simulations [Ert95], and timing data with restricted forms of stack caching: Gforth was only able to perform single-state stack caching with one register, and static stack caching with the canonical state containing 0 or 1 registers [EG04].

In this paper, we describe how we lifted these restrictions (Section 3), and present empirical results, including timing results for several different machines (Section 4).

2 Background

This section gives an overview of stack caching [Ert95, EG04].

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

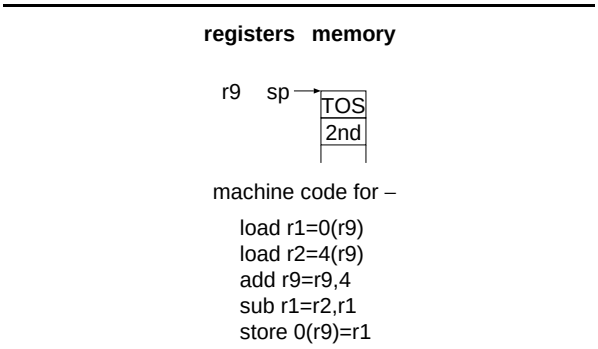


Figure 1: A straight-forward representation of the stack

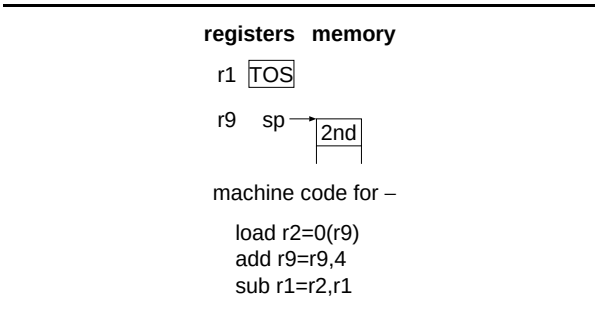


Figure 2: Keeping the top-of-stack in a register

2.1 Stack representation

A straight-forward representation of the stack is to keep all stack items in memory, and have a stack pointer that points to the top-of-stack (Fig. 1). This requires a memory access for every stack accessed stack item.

A frequently used improvement over the straight-forward representation is to keep the top-of-stack in a register (Fig. 2). This makes the frequent accesses to the top-of-stack substantially cheaper.

2.2 Using several registers

One might consider keeping more stack items in registers all the time. However, this does not necessarily lead to an improvement in running time, because with many stack items in registers, changing the stack depth often requires additional moves between registers (Fig. 3). Whether more stack items provide a speedup, depends on the mix of primitives,

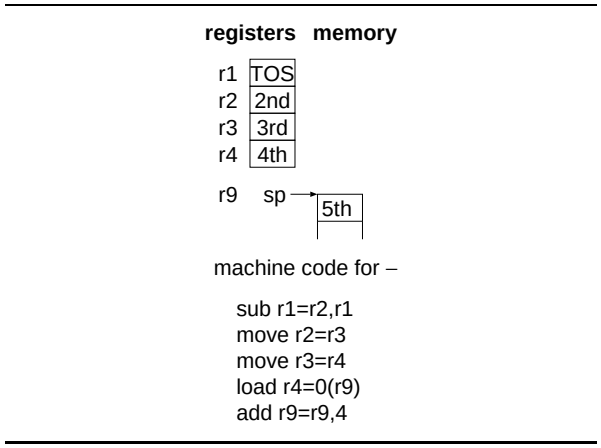


Figure 3: Keeping the four top stack items in registers

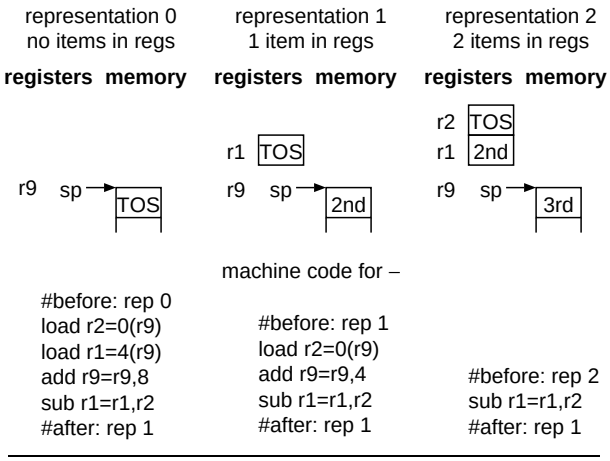


Figure 4: A stack cache with multiple stack representations

and on the characteristics of the machine executing the code.

In the past we have presented only simulation results for stack caches with more than one register. In this work we present timing results from real machines.

2.3 Multiple stack representations

To avoid the cost of the register moves (and other costs) when changing the stack depth, one could change the stack representation during the execution of a primitive (Fig. 4); note how cheap `-` becomes when it starts in representation 2 and is allowed to finish in representation 1. Of course, then the next primitive executed has to be in a version that starts in representation 1 (or we have to insert additional code that switches between representations). So, in order to make profitable use of this, we need different implementations of at least the common primitives, for different stack representations.

2.4 Static stack caching

How do we get the right version of the primitive to execute? There are at least two ways, but the more promising one is static stack caching: The compiler keeps track of the stack representation, and for each primitive it has to compile, it compilers an appropriate version of the primitive.

This approach requires that the stack representation is the same when two control flow paths join. Moreover, for simplicity in the compiler it is best if the stack representation is the same at all points where control flow can happen (in compiler terminology, at all *basic block* boundaries); this representation is the *canonical stack representation*.

In earlier work, we only had simulation results for static stack caching [Ert95], or timing results where the canonical stack representation could have at most one register [EG04]. In the present work, we present timing results for stack caches with other canonical states.

3 Implementation

3.1 Interpreter generator

The code for Gforth's primitives is written in a mixture of Forth and C. E.g., here is the code for the primitive `+`:

```
+ ( n1 n2 -- n ) core plus
n = n1+n2;
```

An interpreter generator [EGKP02] translates this code into (GNU) C code, and gcc then translates it into an executable interpreter.

One important aspect of the interpreter generatore is that it generates all the stack access code for a primitive from the specification of the stack effect in the first line of the primitive's specification.

To implement stack caching, we generalized the access-generating code to deal with arbitrary stack representations, including different representations before and after the primitive. We also added ways to specify stack representations, and to determine which versions of a primitive are generated.

One problem in this context were primitives that access a stack pointer explicitly in their C code, either because they have to manipulate it (e.g. `sp!`), or because they do something beyond the descriptive powers of the stack effect specification in the interpreter generator (e.g., `?dup`). The primitives affected in Gforth are: `sp@ sp! fp@ fp! ?dup ?dup-?branch ?dup-0=-?branch pick >float fpick` and some C call interface primitives.

In our first foray into multi-state stack caching [EG04], we just left these primitives alone, so that they would just keep working with 0 or 1 stack items

in registers. However, this restricted the canonical stack representations we could use to just those with 0 or 1 stack items in registers.

In the present work, we eliminated this restriction: You can now put the string `...` (possibly prefixed by a stack prefix) into the stack effect description of a primitive; this causes the generator to flush all the cached stack items to memory and let the stack pointer(s) point to the top-of-stack, thus presenting the C code with the straightforward stack representation (Fig. 1); after the C code, stack items are loaded into registers and the stack pointer is adjusted as is necessary for the representation after the primitive. Here is an example:

```
pick ( S:... u -- S:... w ) core-ext
w = sp[u];
```

Here the `S:...` indicates that the data stack has to be flushed before and reloaded after the primitive. An additional advantage of this approach is that these primitives became much easier to understand than they used to be; before this extension, one had to consider the kind of code that the generator would produce, often with conditional compilation for dealing with the differences between using 0 or 1 register.

3.2 Code generator

When Gforth compiles Forth code (or loads the system image), it has to select which versions of the primitives (out of several with different input and output stack representation) should be used. This selection is performed by C code that hooks into the Forth compiler via `compile,` and is also called from the loader. This code generator uses a shortest-path algorithm for selecting the optimal sequence of primitive versions (optimality criterion: minimum sum of the native-code sizes of the primitive versions). This code generation process is described in more detail in our earlier work [EG04].

3.3 Effects on Forth code

To work correctly with stack caching, the colon definitions must not access stack items in memory (with `sp@` and memory operations). Fortunately, there was only one colon definition in the Gforth system that did this: `roll`. This definition was changed into one that does not use `sp@` and does not use memory operations to access stack items.

In addition to that, there were some very small changes to make the static stack caching code generator (written in C) aware of control flow joins (`then`, `begin`).

These were the only changes that were needed in the Forth code of the Gforth system, so the changes for static stack caching were fairly local.

3.4 GCC issues

Stack caching introduces additional versions of the primitives. The versions of Gforth we used for the present work contain around 1200 primitives and their versions: 355 basic primitives (starting and ending in the canonical state), 795–848 versions of popular primitives for other transitions between stack representations, and 13 superinstructions (deactivated in our experiments).

In older versions of GCC and with our old way of coding NEXT in the primitives, having so many primitives and their versions resulted in a huge memory consumption (several hundred MB) and long compile times (on the order of a half-hour).

With more recent GCC versions, this problem was not present, but they generated code that disabled dynamic superinstructions, a very profitable optimization in Gforth that is also essential for our implementation of static stack caching.

We worked around both of these problems by changing the way we code NEXT. Instead of appending the NEXT sequence including an indirect `goto` (`goto *`) to each primitive, we just have one indirect goto (very early) in the whole function. At the end of each NEXT, we append a direct goto to this indirect goto:

```
engine(...)
{
   ...
  before_goto:
   goto *real_ca; /* indirect goto */
  after_goto:
   ...
  I_plus:
   ... /* all of + except NEXT */
   ip++; /* maintain ip for accessing
            immediate arguments */
  K_plus:
   real_ca = ip[-1]; /* NEXT, part 2 */
  J_plus:
   goto before_goto;
  ... /* other primitives */
}
```

For dynamic superinstructions, when we want to generate the code for a `+` without a NEXT, we copy the code between `I_plus` and `K_plus` to the native-code area of the current definition. But if we want to include the NEXT (normally that only happens for branching primitives), we copy the code between `I_plus` and `J_plus`, and append the code between `before_goto` and `after_goto`; this avoids the problems with the non-relocatability of the `goto before_goto`.

The benefit of this workaround in our context is that even older gccs compile `gforth-fast` with the 1200 primitive versions in around a minute (on a

1066MHz PPC7447A), using about 50MB of RAM. With newer gcc versions we get engines where dynamic superinstructions work.

The downside of this workaround is that, if dynamic superinstructions are disabled for some reason, the the Forth system runs significantly slower than the old version of Gforth would run when compiled with an older version of gcc: The additional direct branch per primitive costs time; and on CPUs with branch target buffers (e.g., various Pentiums and Athlons), the shared indirect branch has significantly worse branch prediction than the separate indirect branches had. However, ideally dynamic superinstructions are enabled in all situations where performance is important, so this disadvantage should not be a problem.

4 Results

4.1 Hardware

The main component that determines the performance in our benchmarks is the CPU. We used three different hardware platforms with different CPUs: a 450 MHz PPC7400 (PowerMac G4), a 1066MHz PPC7447A (iBook G4), and a 2000MHz PPC970 (PowerMac G5). The PPC7400 is a shallowly pipelined CPU (4 stages in the integer pipeline) that can issue up to two instructions per cycle; the PPC7447A is a deeper (7 stages) and wider (triple-issue) CPU; and the PPC970 is very deep (16 stages) and very wide (five-issue).

So we can expect to see some performance differences from these CPUs, even though they have the same architecture. We use the PPC architecture for our experiments, because gcc is able to allocate many registers for the stack cache on this architecture, unlike on other architectures we have tried (Alpha, MIPS, AMD64, ARM); we believe that this is caused by the much higher number of callee-saved registers in the PPC calling convention compared to other calling conventions.

All of these machines were running Linux, and we benchmarked the same executable programs on all of them.

4.2 Forth systems

We built nine Gforth engines, all of them with 8 registers usable for stack caches. The engines differ in the canonical stack representation they support, one for each number of registers (0–8). The other stack representations can be controlled using a command-line parameter. E.g., we ran the engine built for the canonical state with three registers with just one stack state to get results for single-representation stack caching with three registers. We also ran it restricted to the representations with 0–3 registers with the three-register representation being canonical; similarly for 0–4 registers up to 0–8 registers. In this way the 9 basic engines were used for evaluating 53 stack caching organizations.

Program	Vers.	Lines	Description
cross	0.6.9	3793	Forth cross-compiler
tscp	0.4	1625	chess
brainless	0.0.2	3519	chess
vmgen	0.6.9	2641	interpreter generator
bench-gc	1.1	1150	garbage collector
CD16sim	1.1	937	CPU emulator
brew	t_38	31401	evolutionary playground
pentomino		516	puzzle solver
sieve		23	prime counting
bubble		74	bubble sort
matrix		55	integer matrix multiply
fib		10	double-recursive function

Figure 5: Benchmark programs used

Even though we built the engines with a few static superinstructions, we disabled them in benchmarking, because the combination of static stack caching and static superinstruction is not supported yet, so the static superinstructions just work in the canonical state, and enabling them might suppress some of the effects of stack caching (to a greater extent than an proper combination of stack caching and static superinstructions would).

The engines were built with gcc-4.0.1 (Debian 4.0.1-2).[1]

4.3 Benchmarks

Figure 5 shows the benchmarks we used for our experiments. In addition to timing results, we also present instruction, load, and store counts; they were collected using the performance monitoring counters of the PPC7447A and the `perfex` utility of the perfctr patch for Linux. We use the same executables on all machines, so the number of executed instructions, loads, and stores are the same on all of them. We ran each benchmark three times for each configuration, and present the median of the three runs.

4.4 Run-time and instructions

Figure 6 shows the number of instructions executed by the benchmark *Brainless*. The line labeled `n`

[1] We suspected that auto-increment load and store instructions combined with the selection of which stack item the stack pointer points to might influence the results, so we also performed experiments with compiling with the `-mno-update` flag, which suppresses generating code that uses auto-increments. However, the results were essentially the same either way, so our suspicion was disproved. In this paper, we report the results without `-mno-update`.

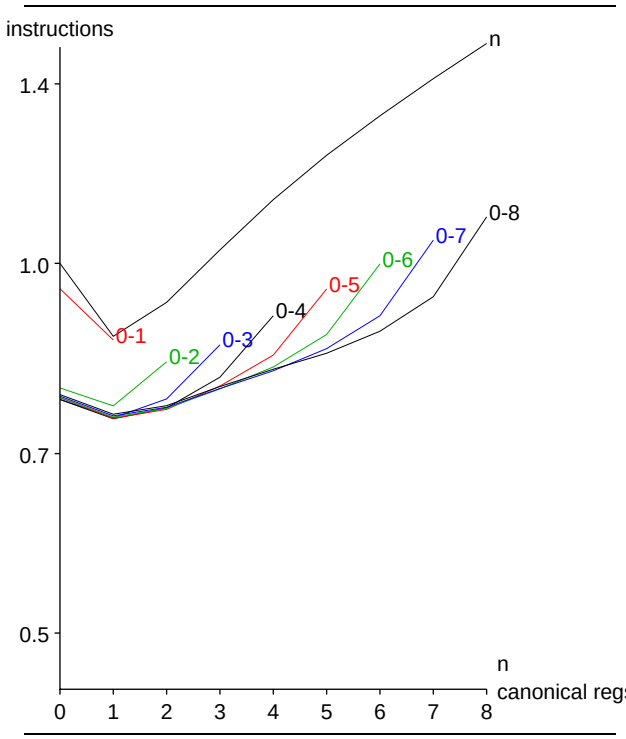


Figure 6: Instructions executed by Brainless

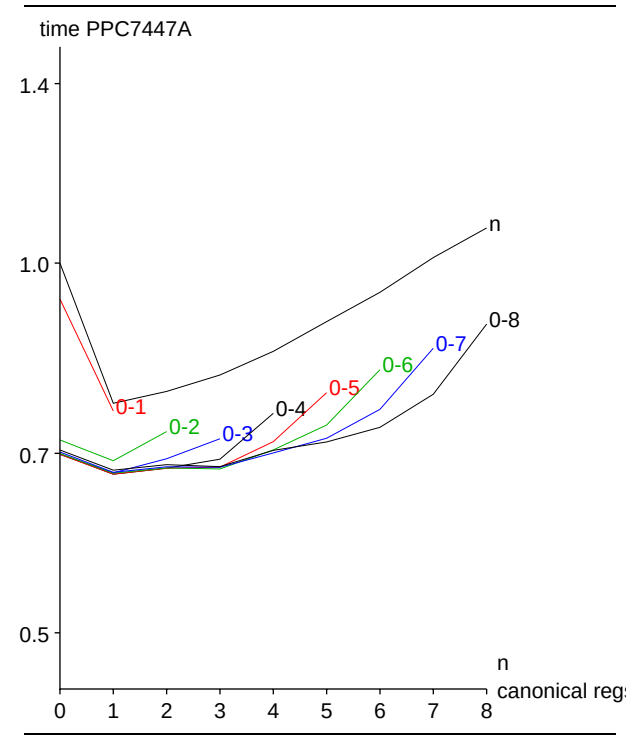


Figure 7: Brainless run-time on PPC7447A

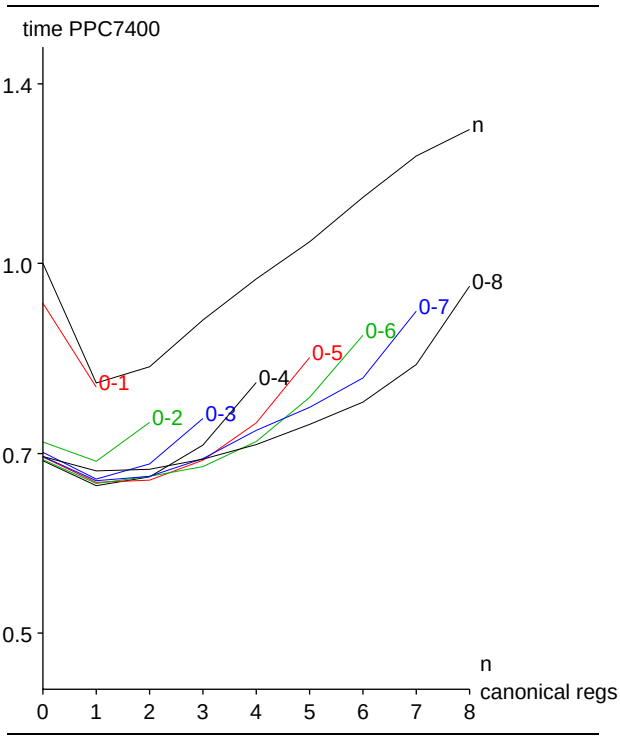


Figure 8: Brainless run-time on PPC7400

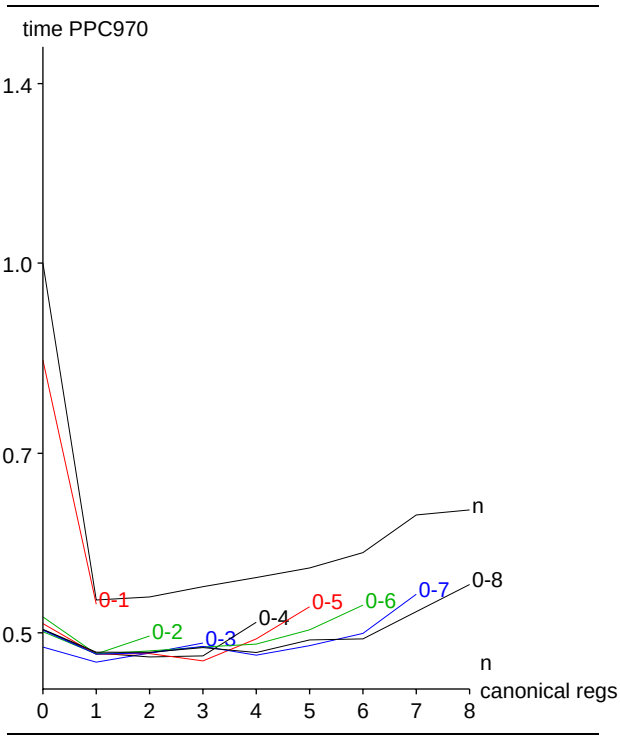


Figure 9: Brainless run-time on PPC970

represents the stack caches with a single stack representation; that stack representation is indicated by the position on the x-axis. The lines labeled 0–x represent stack caches using stack representations with 0 to x registers; the canonical representation is indicated by the position on the x-axis.

Figure 7, 8 and 9 show timing results for Brainless on different CPUs.

Figure 12, 13, 14 and 15 show instruction counts and timing results for all benchmarks; two single-representation results are shown per benchmark: for keeping one stack item in a register all the time, and the best single-representation scheme for the benchmark (this may be different from the best scheme for other benchmarks). Similarly, for the multiple-state schemes the scheme with up to three registers (0-3) with the canonical representation keeping one stack item in a register is shown, and the best multi-representation scheme for the benchmark.

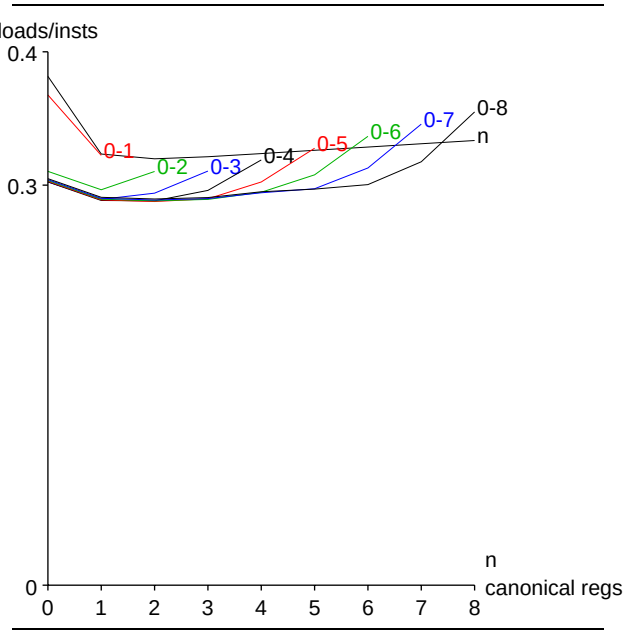


Figure 10: Load instructions executed dynamically by Brainless

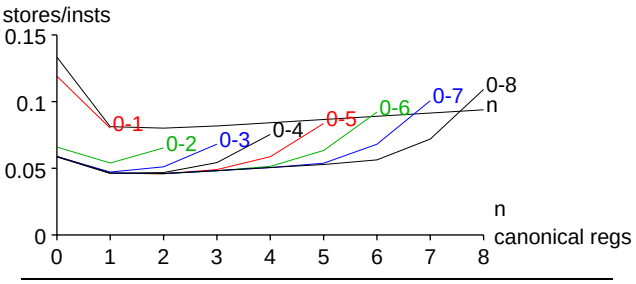


Figure 11: Store instructions executed dynamically by Brainless

Which canonical representation?

For the multiple-representation stack caches, once the number of registers available exceeds those in the canonical representation by two or more, all caches with the same canonical representation perform about the same. The number of instructions executed is smallest for the canonical stack representation with one register (except for some of the smaller benchmarks). Similarly, for the single-representation stack caches, the one with one register executes the least instructions.

The PPC7400 timings behave quite similar to the instruction counts, although the timing reduction is somewhat higher than the instruction reduction; on the PPC7447A and especially the PPC970 the times for canonical representations with more than one registers rise much more slowly (and sometimes not at all).

Nevertheless, even on those CPUs using the one-register representation as canonical representation or, for single-representation stack caches, as the representation is optimal for many benchmarks, and close to optimal on the others.

How many registers?

With the canonical representation set to using one register, how many registers should be used for a multiple-representation stack cache? More than three registers does not help much (see Section 5 for an explanation); so if three registers are available, they should be used. Two registers are almost as good, but with just one register, the speedup over the one-register single-representation stack cache is tiny.

Are multiple representations worthwhile?

In the setup we evaluated, the 0-3 stack cache with the one-register canonical representation provides up to a factor of 1.53 speedup (Pentomino on the PPC7400) over the single-representation stack cache with one register. If enough registers are available (at least two), the speedup may well be worth the implementation cost.

Benefit of single representation

While the case for multiple stack representations depends on the circumstances, the case for keeping one stack item in registers all the time is pretty clear. For a tiny increase in implementation complexity we get a significant increase in performance, in particular on the PPC970. In earlier work [EGKP02] we have also tested this on other CPUs; the results were not as spectacular as for the PPCs, but still worthwhile.

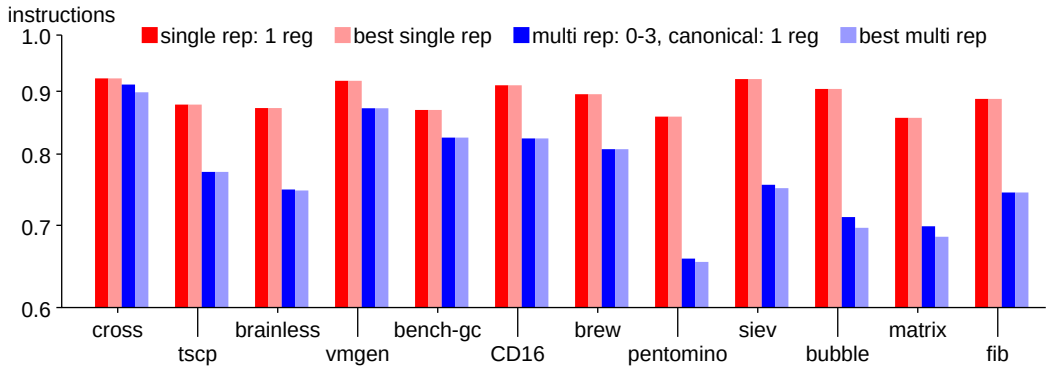


Figure 12: Instructions executed dynamically relative to the straight-forward stack representation

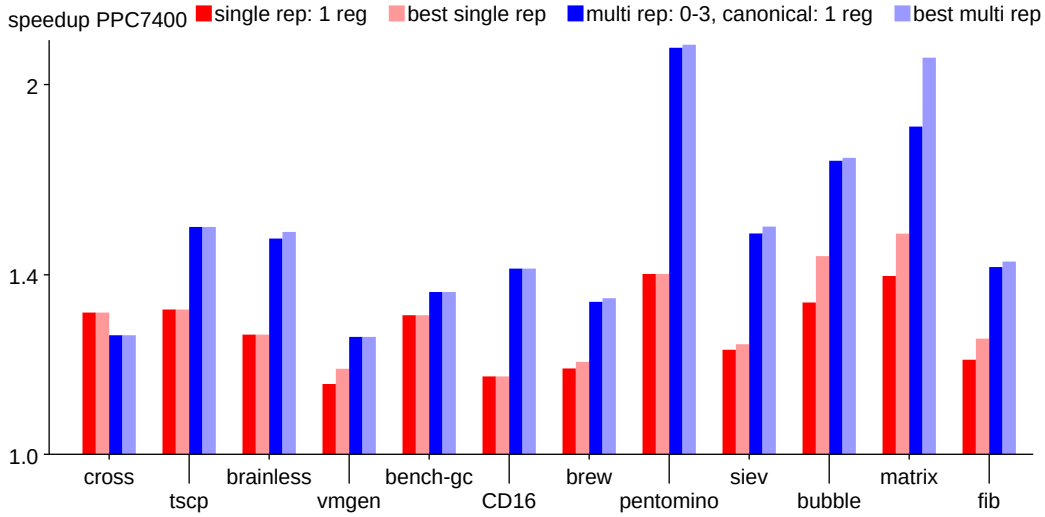


Figure 13: Speedup on the PPC7400 over the straight-forward stack representation

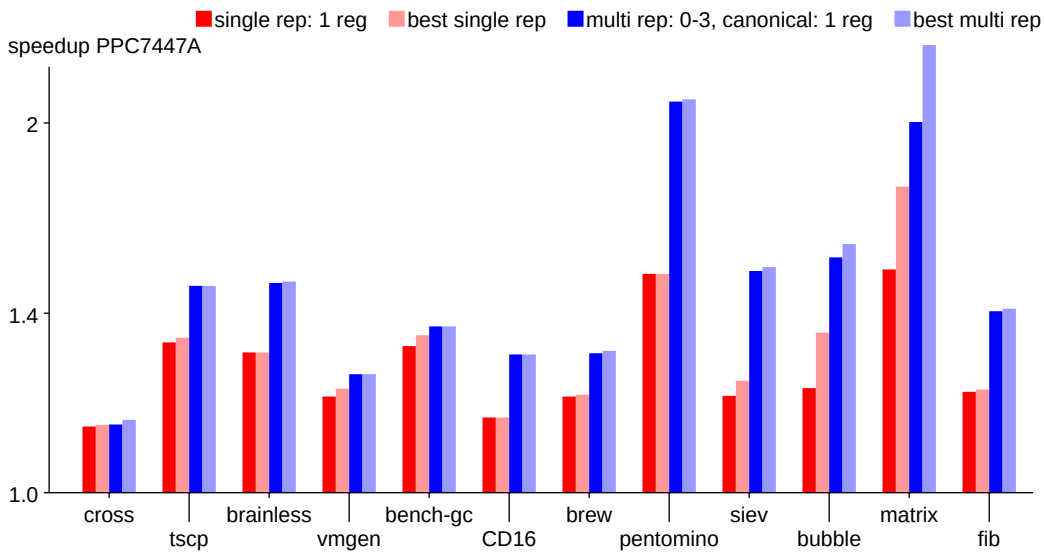


Figure 14: Speedup on the PPC7447A over the straight-forward stack representation

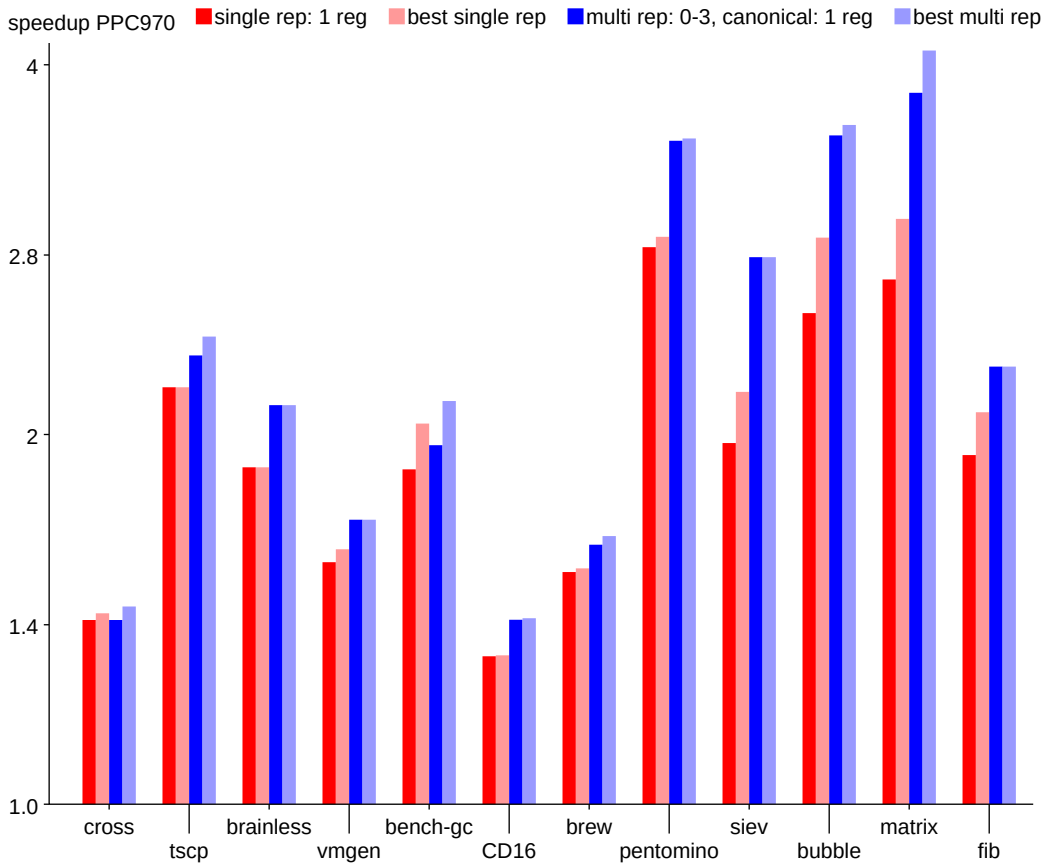


Figure 15: Speedup on the PPC970 over the straight-forward stack representation

Loads and Stores

Figure 10 and Fig. 11 shows the number of executed loads and stores as proportion of the number of executed instructions.

Stack caching reduces both loads and stores by about the same number. However, there is a big baseline of loads that do not perform stack accesses, which is the reason for the difference in the way the pictures look.

One big contributor to this baseline is that each primitive still loads the address of the next one. This is mostly redundant in the context of dynamic superinstructions and could be optimized away.

The significance in the number of loads and stores is that some CPUs have particular performance issues related to these instructions. In particular, there are a number of CPUs that are store-limited, because their writes go off-chip (no on-chip caches, or only write-through on-chip caches); CPUs of this class are the 486DX2 and some 486DX4s, the MicroSPARC II, the 21064 and the 21164PC; for newer high-performance CPUs this is a problem of the past, but it might show up in embedded systems (and sometimes as a bug workaround elsewhere). For store-limited CPUs the speedup can be directly proportional to the reduction in stores.

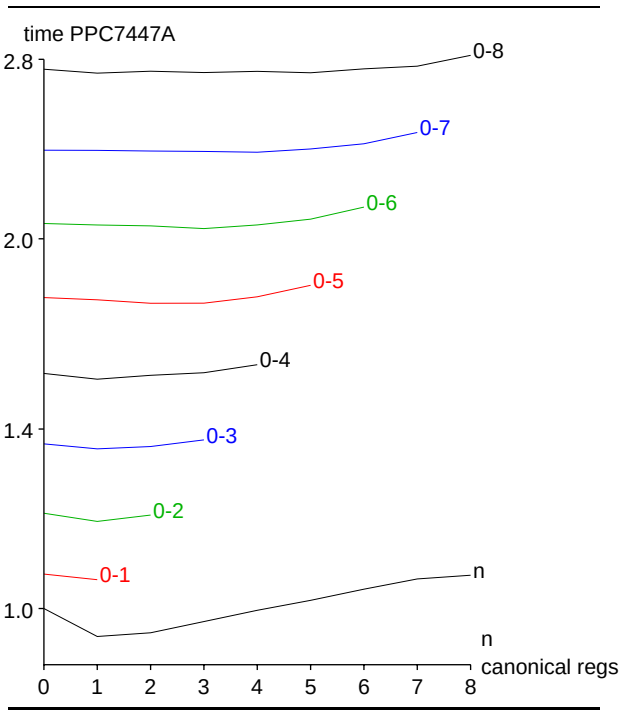


Figure 16: Gforth startup time

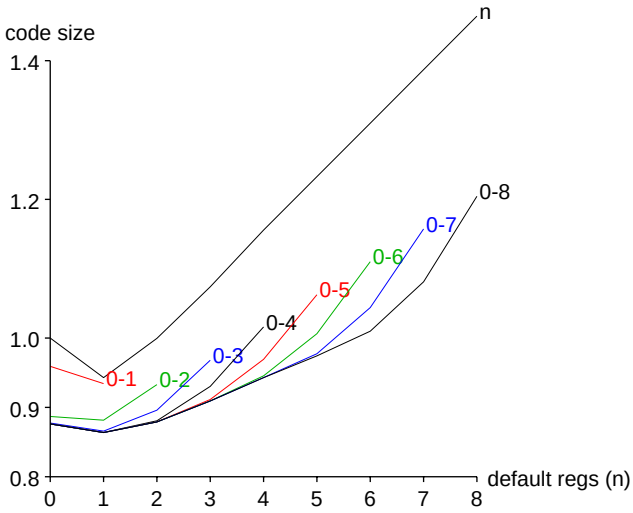


Figure 17: Code size of the dynamically generated native code for the Gforth image

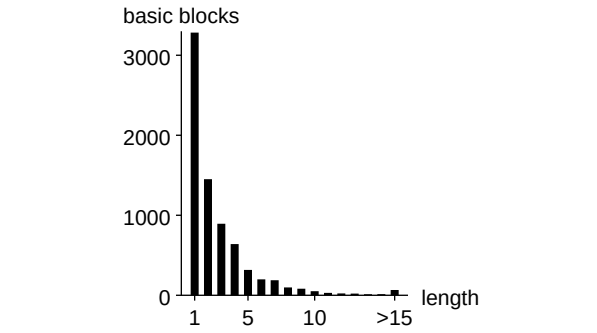


Figure 18: Number of primitives per basic block (static) for Brainless

4.5 Compile time

The time taken by the shortest-path algorithm used in the code generator (Section 3.2) takes time linear with the number of stack representations. This affects the startup time of Gforth (where the code generator is applied to the code of the image file), and the compilation speed. Figure 16 shows the resulting changes in the startup time. Note that even with 9 representations, the startup time of Gforth on the 1066MHz PPC7447A is still only 0.05s, so in most applications this is not a serious problem; however, it is visible in the results of short-running benchmarks.

It is possible to have a faster code generator that uses a two-pass automaton and has performance independent of the number of stack representations, but we have not implemented that (yet).

4.6 Code size

The code size is also affected by stack caching (Fig. 17). With a single stack representation with one register, the code is 0.94 times as large as without stack caching. With multiple representations the code size can be reduced to 0.86 times the size without stack caching.

However, the additional primitive versions necessary to make multiple representations effective also should be added to the code size; for the engine with 0–8 registers, with one register for the canonical representation, the additional code size is 26068 bytes for the primitives alone; the additional code size for 0–3 registers would be significantly smaller, probably around 10KB. For the Gforth image alone going from always-1 to 0–3 registers saves 24024 bytes, so multiple-representation stack caching can pay for itself already before compiling any additional code.

On the other hand, at least Gforth needs another copy of the additional primitives (for determining relocatability), plus embedded padding, plus some tables describing the additional primitives. And for a smaller image, the savings would be smaller. So multi-representation stack caching does not necessarily reduce the code size.

Moreover, if code size is at a premium, the user would not use dynamic superinstructions, and there would be no code size savings from multiple representations, only the cost of the additional primitives.

5 Further work

The improvement of multiple-representation stack caching over single-representation stack caching is a little disappointing. One reason for this could be that the basic blocks in Forth code are very short, forcing a return to the canonical representation very often (Fig. 18). In particular, for the large number (45% for Brainless) of basic blocks with length one there is no difference between a multiple-representation stack cache and a single-representation stack cache. So, given this basic block length distribution, it is not very surprising that there is not that much performance difference between single and multiple representations.

So if we apply optimizations that make the basic blocks longer, we might see quite different results than those in this paper. For Forth the most promising of these optimizations is inlining [GE04]. We will investigate the effect of inlining in the future.

6 Related Work

Stack caching was first published by DeBaere and Van Campenhout [DV90], who presented a small example of dynamic stack caching.

Ertl [Ert95] discussed stack caching in more detail, including various stack cache organizations, static and dynamic stack caching, and presented results in numbers of eliminated loads, stores, and stack pointer updates, but produced no full implementation.

Sun's Hot Spot JVM system performs dynamic stack caching in its interpreter part [Gri01]: It caches up to one stack item in registers; for each of the four types (int, long, float, double), it has a separate state that represents the presence of one stack item of this type in registers (different registers are used for some of these types). It is not necessary to implement instances of all instructions for all states, because the type rules of the JVM disallow many state/instruction combinations.

Ogata et al. [OKN02] implemented dynamic stack caching with up to two registers, but eventually dropped it because the speedup from that on their Power3 machine was not large enough (1%–4% over single-state stack caching) to justify the complexity.

The differences between the present paper and these papers is that we present an implementation of *static* stack caching.

Peng et al. [PWL04] introduce a technique for saving real-machine code space in static stack caching (with an unconventional stack cache organization) by arranging the code for the VM instruction instances such that they share one piece of code, with different entry points for the different instances. The difference between this paper and our work is that we combine static stack caching with dynamic superinstructions and that we use different and more stack cache organizations (designed for execution speed, not code sharing).

In our earlier work [EG04], we already combined static stack caching with dynamic superinstructions. In this work we expand on that work by implementing stack caching with arbitrary canonical representations, and evaluating the resulting stack cache organizations. We also discuss issues related to high-level Forth code and some issues we had with gcc and how we solved them; also, in the present paper we only give an overview over the code generation topics that were discussed in depth in our earlier papers.

7 Conclusion

For single-representation stack caching, keeping one stack item (the top-of-stack) in a register is usually optimal; the resulting speedup (over using the straight-forward stack representation) depends on the benchmark and the CPU, and can reach up to a factor of 2.84 (pentomino on PPC970); however, on most other CPUs the speedups are significantly smaller.

For multiple-representation stack caching, using a canonical state with one register is often optimal; with that fixed, using more than three registers for the stack cache provides little benefit. This stack cache organization provides speedups of up to a factor 3.80 (matrix on PPC970), but again the results on other CPUs and other benchmarks are often considerably less. The speedup of using this stack caching scheme over single-stack stack caching can reach up to a factor of 1.53 (pentomino on PPC7400). Optimizations that make basic blocks longer (e.g., inlining) might change these results.

References

[DV90] Eddy H. Debaere and Jan M. Van Campenhout. *Interpretation and Instruction Path Coprocessing*. The MIT Press, 1990.

[EG04] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *IVME '04 Proceedings*, pages 7–14, 2004.

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

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

[Gri01] Robert Griesemer. Interpreter generation and implementation utilizing interpreter states and register caching. Patent 6192516 B1, US, 2001.

[OKN02] Kazunori Ogata, Hideaki Komatsu, and Toshio Nakatani. Bytecode fetch optimization for a Java interpreter. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 58–67, 2002.

[PWL04] Jinzhan Peng, Gansha Wu, and Guei-Yuan Lueh. Code sharing among states for stack-caching interpreter. In *IVME '04 Proceedings*, pages 15–22, 2004.

Xchars
or
Unicode in Forth

First Experiences

M. Anton Ertl*
TU Wien

Bernd Paysan

Abstract

When dealing with different scripts at the same time (e.g., Latin, Greek, Cyrillic), or with Chinese ideograms, 8-bit fixed-width characters are too narrow. However, many Forth programs have an environmental dependency on `1 chars = 1`, so just making Forth characters wider would cause quite a lot of portability problems. We propose to add xchars for dealing with potentially wider, variable-width characters. This extension is relatively painless, requiring changes in only those program parts that work with individual characters, if they should work with the extended characters; uses of string words need no changes to work with extended characters. The xchar words can also be implemented on 8-bit-only Forth systems, so programs written to use xchars can also work on such systems.

1 Introduction

Most Forth systems today support character sets fitting into 8 bits, such as ASCII (7 bits) and its 8-bit extensions like ISO Latin-1.

However, such 8-bit character sets are not sufficient to support Chinese, Japanese, and Korean Han ideographs, or to express a text that contains, say, German, Russian, *and* Greek words. To address this problem, Unicode[1] was developed. Unicode is a universal character set.

There are several alternative encodings of Unicode characters: In UTF-32 each character consists of 32 bits, in UTF-16 each character consists of 1–2 16-bit entities, in UTF-8 each character consists of 1–4 8-bit entities. I.e., UTF-8 and UTF-16 are variable-width encodings.

How can Forth accomodate Unicode? ANS Forth only allows ASCII or only graphic ASCII characters in many contexts. However, ANS Forth also supports fixed-width, but large characters; e.g., a Forth system could use 32-bit characters to support the UTF-32 encoding of Unicode; since the codes for the ASCII characters are the same in Unicode, this would actually be a fully compliant ANS Forth implementation. Indeed, Jax4th was one of the first ANS Forth implementations and implemented characters as fixed-width 16-bit characters (for the then-current 16-bit (subset of) Unicode).

However, most Forth programs, even if they are otherwise mostly ANS Forth compliant, assume that `1 chars` produces 1.[2] These Forth programs would not work correctly on a system where `1 chars` produces 4, as would be the case with UTF-32 characters on a byte-addressed machine. And it is relatively hard to find all the places where one forgot to insert `chars` or `1 chars /` or where one used `1+` instead of `char+`. So going to UTF-32 characters would be a rather painful option.

Fortunately, Forth programs usually do not work with individual characters (with, e.g., words like `emit`) in many places. They work much more often with strings of characters (with, e.g., words like `type`). So if we find a way to deal with Unicode where string-handling code would continue to work, and only character-handling code needed changing, that solution would require much less porting effort for most programs than using UTF-32 with the existing character words and an appropriate `chars` size.

In this paper, we propose such a solution based on a new character type (xchars) and words for dealing with that type. In the following paper, we explain and discuss the new data types and words (Section 2), look at scenarios for using various en-

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

[1] Actually, there were two standards: ISO 10646 and Unicode, produced by two different organizations, resulting in two standards documents; fortunately, the two documents define the same character set. We use the name Unicode throughout this paper.

[2] Since all widely used ANS Forth systems have the property that `1 CHARS` produces 1, it is pretty much impossible to test that a program does not have this environmental dependency.

codings and character sizes in Forth systems (Section 3), give some examples of using these words (Section 4), report our experience with implementing these ideas in Gforth (Section 5), and compare our work to that of others.

2 Glossary

The following set of words is not final. It contains a number of redundant words, and we might decide to recommend a smaller set for widespread adoption. Conversely, there might also be words that are useful and that we missed or are still undecided (see Section 2.4).

If you are missing string words (like `type`), that's because you can use the ANS Forth string words on strings containing xchars.

2.1 Data types

xc An xchar (extended character) on the stack. For Unicode characters this will typically be the (decoded) Unicode number of the character.

xc-addr The address of an xchar in memory. Xchar addresses are character-aligned. An xchar can be represented (encoded) in memory differently than on the stack.

xc-addr u A string containing xchars. *u* counts the chars, not the xchars (or tha aus) in the string. All ANS Forth string words can be used on such a string.

2.2 Words

xchar+ (xc-addr1 – xc-addr2) Corresponds to `char+`. xc-addr2 is the address of the xc after xc-addr1.

xchar- (xc-addr1 – xc-addr2) Corresponds to `char-`. xc-addr2 is the address of the xc before xc-addr1.

+x/string (xc-addr1 u1 – xc-addr2 u2) Corresponds to `1 /string`.

-x/string (xc-addr1 u1 – xc-addr2 u2) Corresponds to `-1 /string`.

xc@ (xc-addr – xc) Corresponds to `c@`. Fetch the xchar at xc-addr onto the stack.

xc@+ (xc-addr1 – xc-addr2 xc) Fetch xc from xc-addr1; xc-addr2 is the address of the next xchar.

xc@+/string (xc-addr1 u1 – xc-addr2 u2 xc) Fetch xc from xc-addr1 and also perform the action of `+x/string`.

xc!+? (xc c-addr1 u1 – c-addr2 u2 f) If the buffer at c-addr1 u1 is big enough for xc, store xc there, f is true and c-addr2 u2 describe the rest of the buffer. If the buffer is too small, f is false and c-addr2 u2 is the same as c-addr1 u1.[3]

xc-size (xc – u) U is the number of chars that xc takes when stored in memory.

-trailing-garbage (c-addr u1 – c-addr u2) Given a string c-addr1 u1 containing xchars and further chars that do not form a complete xchar, c-addr u2 is the same string with only the complete xchars.

wcwidth (xc – u) U is the display width of xc on a monospaced display. Currently this word can produce the values 0, 1, 2.

display-width (xc-addr u – u2) u2 is the display width of the string xc-addr u on a monospaced display.

Ambiguous conditions exist, if the xchar(s) read from memory by `xchar+ xchar-+x/string -x/string xc@ xc@+ xc@+/string display-width` are not properly encoded xchars[4], or if the count would underflow (for `+x/string xc@+/string -trailing-garbage`).

In addition, words like `key`, `emit`, `char` and `[char]` have to be extended to work with xchars.

2.3 Requirements and Guarantees

An encoding to be used with the xchar words must have the following properties:

1. The length of an xchar can be determined in forward processing (every encoding has that property).
2. The length of an xchar can be determined in backwards processing (not every encoding has this property, but UTF-8, UTF-16, some encodings for Asian languages, and of course fixed-width encodings have it).
3. Partial xchars can be recognized (this is usually a consequence of backwards processability).
4. The on-stack representation of ASCII characters is the ASCII number (so that `char`, `emit` etc. work as expected).

[3] Bernd Paysan's reference implementation also cantains a word `xc!+ ( xc xc-addr1 -- xc-addr2 )`, but this word is cumbersome to use safely and easy to use not safely, leading to buffer overflows (like C's `strcat()`).

[4] E.g., in UTF-8, if an ASCII character is followed by a character in the range $80–$bf, or if the xchar is not encoded in the shortest possible sequence.

In addition, the following property is needed to ensure that all ANS Forth programs work on a system with xchars when processing ASCII-only strings (which is the only case that ANS Forth actually covers):

5. The in-memory encoding of ASCII characters is the same for xchars and chars.

2.4 Input and output

The xchars words were designed for having one universal encoding used throughout the Forth system. However, Forth code might have to deal with other encodings on I/O.

For I/O of text files (and other things supported by the Forth system with a file-like interface) in a specific encoding, the encoding of the external text could be specified in the fam (file access mode) parameter of `open-file` (with a `bin`-like word), and the reading and writing words would perform the conversion between the external and the internal encoding. One consequence of this conversion is that you usually cannot use file positions for such files in calculations to compute other file positions (because the size of the data in the file has little relation to the size of the data in the Forth system).

For text fields in binary files (e.g., Java `.class` files), the file has to be read/written in binary mode, and the text fields have to be converted between the external and the internal encoding with string conversion words.

These ideas have not been implemented in Gforth yet, and there are no word specifications yet.

2.5 Multiple internal encodings

Some people have suggested words for changing the Forth-internal encoding at run-time. We did not design xchars for such an environment, and would probably design an extension for such an environment differently. The way to deal with different encodings in the outside world in the xchars context is to convert them all to a universal encoding in the Forth system, and convert back on output.

The technical problem with switching between the internal encodings is that existing strings will continue to be in the old encoding, and interpreting them in the context of the new encoding will produce wrong results. So the program would have to keep track of which strings are in which encodings and always switch around between encodings, which is cumbersome and error-prone. And if two strings containing different encodings have to be used in the same operation (e.g., in `compare`), there is no way to set the switch right (and actually, with our xchars proposal `compare` does not encoding-dependent work).

3 Implementation scenarios

3.1 8-bit xchars and 8-bit chars

That is very easy to implement on top of current systems. It may appear pointless, but it allows to run code that uses xchars on systems that only deal with 8-bit characters. And it allows developing code on such systems that should work on systems with more featureful xchar implementations (although one should probably still test on a more featureful system). Gforth implements this scenario.

3.2 UTF-8 xchars and 8-bit chars

This combination satisfies all the requirements above (including requirement 5), as well as satisfying the widespread environmental dependency on `1 chars` = 1 (on byte-addressable machines). Moreover, the memory representation of a non-ASCII xchar consists only of non-ASCII chars; this means that even some programs working on individual characters will work on strings containing non-ASCII xchars, e.g., if the program searches for an ASCII character. Gforth implements this scenario.

3.3 UTF-32 xchars and 32-bit chars

This scenario satisfies all the requirements above (including requirement 5), but (on byte-addressable machines) not the environmental dependency on `1 chars` = 1. Xchars don't make much sense in that scenario, classical ANS Forth characters do everything they do.

3.4 UTF-32 xchars and 8 bit chars

This scenario satisfies all the requirements above except requirement 5; in addition it satisfies the environmental dependency on `1 chars`= 1. While such a system does not conform to ANS Forth (because requirement 5 is not satisfied), it probably takes less effort to port most programs to such a system than to a system like that in Section 3.3.

If this scenario would become the standard scenario, it would make sense to define a different wordset optimized for fixed-width wchars for it rather than our xchars wordset, which is designed for dealing with variable-width characters.

3.5 Other scenarios

Scenarios involving UTF-16 have similar tradeoffs to the UTF-32 scenarios, except that UTF-16 is a variable-width encoding.

4 Code examples

Here we present some examples of using the xchars words.

One thing that we noticed is that it is actually not that easy to find examples where characters are dealt with individually (instead of in strings).

The following word works like `type`, but prints the string back-to-front.

```
: revtype1 ( xc-addr u -- )
    over >r + begin
        dup r@ u> while
            xchar- dup xc@ emit
    repeat
    r> 2drop ;
```

One other thing we noticed is that often, instead of converting an xchar to the on-stack representation, it can just as well be treated as a string (and this is often more efficient):

```
: revtype2 ( xc-addr u -- )
    over >r + begin
        dup r@ u> while
            0 -x/string over swap type
    repeat
    r> 2drop ;
```

Here is another example, implementation of the widely-available word `scan` that searches for a character in a string. First, here is an xchar variant of the non-xchar version in Gforth:

```
: scan1 ( xc-addr1 u1 xc -- xc-addr2 u2 )
    >r
    BEGIN
        dup
    WHILE
        over xc@ r@ <>
    WHILE
        +x/string
    REPEAT  THEN
    rdrop ;
```

And here is a version that deals with the xc as string:

```
: xc->s ( xc -- xc-addr u )
    \ convert xc into ALLOCATEd
    \  in-memory representation
    dup xc-size  dup chars allocate throw
    swap ( xc xc-addr u )
    2dup 2>r xc!+? 0= abort" bug"
    2drop 2r> ;

: scan2 ( xc-addr1 u1 xc -- xc-addr2 u2 )
    xc->s 2dup 2>r search 0= if \ no match
        dup /string then
    2r> drop free throw ;
```

In many cases, the programmer can also provide the xchar as string and call `search` directly instead of through `scan2`.

Finally, here is a primitive implementation of `accept` for xchars.

```
: accept1 ( c-addr +n -- +n2 )
    over >r begin
        key dup #cr <> while ( c-a1 u1 xc )
            dup 2swap xc!+? >r rot r> 0= if
                drop #bell then
            emit
    repeat
    2drop r> - ;
```

5 Experience

We have implemented Xchars and UTF-8 support in the Gforth development version in December 2004 and January 2005, during the course of a month. The xchars addition itself took only a week (after earlier work on an UTF-8 specific wordset).

The main code changes were the addition of a 156-line file for UTF-8 handling, an 80-line file for generic xchar handling and for the 8-bit implementation, changes in `accept` (20 deleted lines, 117 lines added), and changes of less than 100 lines overall in about five other files.

Overall, these changes were relatively painless, and certainly much easier than the changes we would expect had we tried to change the char size.

One interesting challenge was that we did not implement `display-width`, and had to work around that lack in two places:

We use a pretty sophisticated editor for `accept`, where the user can move the cursor back and edit there without deleting the text. In order to achieve this without `display-width`, `accept` now always jumps to the start of the line, draws the part of the line before the cursors, remembers the cursor position, then draws the rest of the line and restores the cursor position to the remembered value.

The other problem was indicating where in an input line an error had happened. Originally Gforth did this by having a second line below the first with `^^^` characters pointing out the word. Now Gforth indicates the word by surrounding it with `>>>` and `<<<`.

Figure 1 gives an idea of how Gforth processing Unicode looks, including a case where an error message is shown.

6 Related work

Jax4th for Windows NT by Jack Woehr was one of the first dpANS Forth systems. It supported

```
Gforth 0.6.2-20030910, Copyright (C) 1995-2004 Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY; for details type `license'
Type `bye' to exit
cr s type
ᚠᛇᚻ ฉันกินกระ לאכול ожy γυαλιά ok
cr s revtype1
άιλαυγ yжо לוכאל ะรกนิกนัฉ ᚻᛇᚠ ok
cr s char כ scan1 type
כול ожy γυαλιά ok
cr pad 10 accept1
ᚻᛇᚠ ok
. 9  ok
pad 9 type ᚻᛇᚠ ok
: לאכול ." ฉันกินกระ" ;  ok
cr לאכול
ฉันกินกระ ok
cr אכול

*the terminal*:9: Undefined word
cr >>>אכול<<<
Backtrace:
$30005FEC throw
$30012418 no.extensions
```

Figure 1: Gforth processes Unicode characters

the then-16-bit Unicode by making characters 16-bit in size, making use of the freedom that ANS Forth had given to Forth systems in this area (by making `1 chars` = 2). However, Jax4th was not used widely, and all widely-used systems implement `1 chars` = 1. More importantly, many near-ANS programs have an environmental dependency on `1 chars` = 1 and would break on systems like Jax4th. Therefore we decided to take a different approach in Gforth and introduced xchars.

Pelc and Knaggs [PK01] identified the same problems as we did (in particular the widespread environmental dependency on `1 chars` = 1), and similar to us propose adding new words for dealing with wider characters: They propose adding wide-character versions of the existing character and string words, for use with wide fixed-width encodings; the system and old-style applications would continue to use the regular character and string words, but applications could be converted to use these wide-character words. In contrast, we propose adding words that support variable-width encodings, but only for words that deal with individual characters; the string words work just as well for strings containing extended characters as for strings containing classical characters. Our approach requires less conversion work, so we propose applying it throughout the system instead of just to application data.

Java uses Unicode as character set and UTF-16 as internal character encoding.

Kuhn compiled an excellent resource on UTF-8 and Unicode [Kuh05], which is highly recommended for anyone having to deal with these issues and contains many links to other documents on the topic.

7 Conclusion

Xchars allow Forth systems to support Unicode (in particular in its UTF-8 encoding) in a relatively compatible way: String words (and programs using them) continue to work without changes; words dealing with individual characters work as usual with ASCII characters, but have to be adapted for working with extended characters.

Xchars can also be implemented easily on systems that only support 8-bit character encodings, so programs using xchars are not restricted to systems with Unicode (or other wide character) support.

Xchars have been implemented in Gforth and bigForth. The Gforth porting experience was relatively painless, requiring adding or changing only a few hundred lines of Forth code.

References

[Kuh05] Markus Kuhn. UTF-8 and Unicode FAQ for Unix/Linux. http://www.cl.cam.ac.uk/~mgk25/unicode.html, 2005.

[PK01] Stephen Pelc and Peter Knaggs. ANS Forth and large characters. http://www.mpeforth.com/arena/i18n.widechar.v7.PDF, 2001.

SuDoku Solver Case Study: from specification to RVM-Forth (part I)

Angel Robert Lynas, Bill Stoddart

October 4, 2005

Abstract

A project is underway to formulate a development cycle from B — suitably augmenting its implementation language B0 with reversibility constructs — to a coded implementation in the reversible target language RVM-Forth[3, 4] with translation schemas defined for this final stage. This paper describes the first phase of a case study using the puzzle SuDoku to investigate possible ways of fleshing out such a development cycle. We adopt an experimental approach, using a relatively simple specification as a springboard for what a generated code implementation might look like, and explore correspondences between the specification and implementation.

1 Introduction

1.1 Background and Terminology

The ancient Japanese puzzle SuDoku is of course no such thing; it's American and only a few decades old. Its basic form is a 9×9 grid of squares (which we refer to as a "board", not entirely defensibly) divided into nine 3×3 subsections, hereinafter called "sectors". A puzzle consists of a partly filled-in board which must be filled in such that every row, column and sector contains all the digits 1 to 9.

The structure of the puzzle and its solutions leads naturally to a set-based model, wherein the constraints on a square's value and the properties of a solution can be readily expressed.

The target programming language is based on ANS Forth, and runs in a Reversible Virtual Machine developed by Bill Stoddart [3, 4] with an embedded set implementation built on work by Frank Zeyda [5]. The language will be referred to as RVM-Forth; the defining feature which concerns us is a guard/ choice set of constructs, described in [2], whereby a non-deterministic reversible choice (written as CHOICE) can be made from elements in a set. The guard construct takes a flag from the stack, and if this is false, reverses to the last non-deterministic choice, choosing another forward path. Should there be no choices left, the previous CHOICE is revisited. On running out of options, a `ko` prompt is given to signal this to the user (for instance, if a puzzle turns out to have no solution).

Variables can be declared as reversible, in which case their earlier values are restored on reversal; thus the (important) state obtaining at the time of the CHOICE can be reinstated. The full code for the implementation is provided in appendix C.

1.2 Objectives

The overall context of our research is to investigate the formal development of reversible programs, using a modified version of the B Method[1]. The B language has an exact mathematical description, so that programs written in it can be subject to formal logical analysis (a theorem prover is an important part of any B development environment). B presents a user with (at least) two levels of the B language, which are respectively a specification language (highly expressive but not implementable) and an implementation language. A developer writes both a specification and implementation of a program, and is obliged to show that the implementation satisfies the specification.

Our aim, over several of these reports, is to produce a complete B development cycle of a simple solver, from abstract B specification through to an RB0 implementation (this being our reversible version of the B implementation language).

The RB0 code will compile to RVM-Forth, and we hope to gain some insights into how to optimise that mapping by seeing how various applications can be programmed in RVM-Forth itself. We are particularly interested in set-based representations of data and automatic backtracking, because both of these lend themselves to the logical analysis which is at the heart of the B method. [3]

2 Data Model

The basic requirement is for a mapping from each assigned square of the board to its value, this being the current board; and for the remaining squares, a mapping to a set of their available values. The squares can be indexed sequentially, or by row-column co-ordinates, the latter proving simpler in most areas. We therefore define XY as the set of integers 0..8, and thence a square is a pair (row, column) of type $XY \times XY$, the cartesian product[1] of XY with itself. For convenience, the type of a square is defined as:

$$SQUARE \mathrel{\widehat{=}} (0..8) \times (0..8)$$

The current board is a function from squares to 1..9 — the integer set defined as $DIGIT$. Initially a partial function, the solution sees it become a total function (and trivially a surjection). So defining a Boolean *solved*, we can specify a variable *board*, beginning thus:

$$board \in SQUARE \nrightarrow DIGIT \wedge$$
$$solved = TRUE \Rightarrow board \in SQUARE \twoheadrightarrow DIGIT \wedge \ldots$$

The specification variable *board* becomes the RVM-Forth variable BOARD. There remains a criterion for validity which each square must fulfil, of course. Complementary to this and used in the RVM-Forth implementation is the function from blank squares to their possible values:

$$POSSIBLE \in SQUARE \nrightarrow \mathbb{P}(DIGIT)$$

2.1 Constraint Zone

A further requirement is the notion of a Constraint Zone for a given square, which is the union of its row, column, and sector. The shaded area in fig 1 around square **S** is its constraint zone. The row and column are simply specified; row 3, for instance, is the set

[1] A brief explanation of set operations used in the paper is given in appendix A.

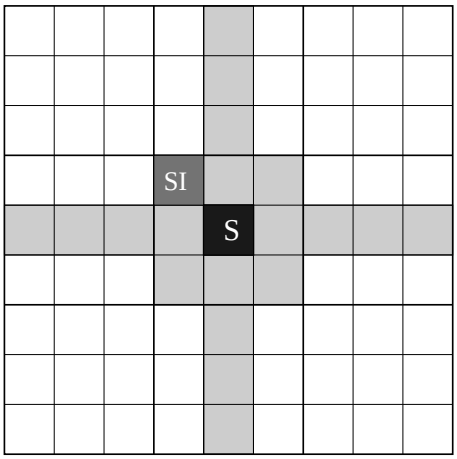


Figure 1: The Constraint Zone for square S

of pairs from (3,0) to (3,8), or the cartesian product of $\{3\}$ and XY. A function can be defined as a constant *row* to encapsulate this, with properties:

$$row \in XY \rightarrow \mathbb{P}(SQUARE) \wedge \\ \forall\, rr.(rr \in XY \Rightarrow row(rr) = \{rr\} \times XY) \tag{1}$$

The sector can be generated in a number of ways; the method used here is to map each square (via a function) to a "sector index" — the square at the top left-hand side of the sector in question (that labelled **SI** in fig 1). The set of sector squares can then be generated orthogonally from these.

Having obtained the constraint zone for a blank square, its relational image with BOARD will yield the subset of *DIGIT* the square *cannot* be assigned. The complement, i.e. *DIGIT* minus these values, will be paired with the square in the initialisation of POSSIBLE. The implementation operation AVAILABLE, which generates these values, is examined below.

2.2 Solution

There remains the rest of the solution specification; while the final board must be a total function, each of its squares must also be valid — that is, the value should not appear elsewhere in its constraint zone. Defining this last as *czone*:

$$czone(r, c) \mathrel{\widehat{=}} row(r) \cup col(c) \cup sector(sectorindex(r, c))$$

A "valid square" function can be defined from a square and a board to a boolean, the salient part of the definition being:

$$is\_valid\_square(rr, cc, bd) = TRUE \Leftrightarrow \\ bd(rr \mapsto cc) \notin bd(\!|\ czone(rr, cc)\ |\!)$$

When *solved* becomes true (having been initialised to false), then for all the squares in *board*, *is_valid_square* is true; it should also be true for every assigned square in partially-filled boards — from which the solution condition would follow. In the code version, this is ensured by the assignment mechanism in any case; illegal assignments would not be possible.

3 Method

An implementation based on the above data model can give us an idea what the final generated product might look like. From the model, an algorithm for a sequential, iterative modus operandi suggests itself; we present the outline followed by some more detailed points. The BOARD and POSSIBLE sets have been initialised at this point: for row r, column c and values v_i, their elements have the forms:

BOARD : $((r, c), v)$
POSSIBLE : $((r, c), \{v_1, v_2, \ldots, v_n\})$

1. Extract set of most constrained blank squares, along with their values (this is a subset of POSSIBLE).
2. Choose (*non-reversibly*) a square from the domain of this set. Remove this entry from POSSIBLE.
3. Choose a value from those available; this must be a reversible CHOICE. Create a pair from the square and value, and add it to BOARD.
4. Update the blanks in the constraint zone of the square, i.e. remove the value just assigned from their available sets.
5. If any of the resulting sets are empty, the board is now non-viable; in this case, we must backtrack to step 3, restoring state along the way.
6. Otherwise, update POSSIBLE itself with these new values.
7. Repeat until board full.

3.1 Algorithm expansion

1. Concentrating on the most constrained blanks simply seems most logical; should a wrong choice be made, there will be fewer subsequent attempts to work through. More sophisticated techniques would be able to reduce the available values by other comparisons with the current state; our naïve version does not apply all possible constraints, but instead allows the backtracking mechanism to take the strain.
2. The above being the case, it's likely that more than one square will be returned, so one must be chosen to work with. A point which bears stressing is that, if a solution exists at all from this stage, it can be found from *any* of these squares. Faster from some than others, perhaps, though there is no way of knowing which. So:
   (a) The choice may as well be random.
   (b) The choice *must not* be reversible.

   The latter point may well not be obvious at first glance (it can be tempting to assume that any choice should be a CHOICE). However, if none of the values from the chosen square lead to a viable board, reversing will simply cause another square from the set (if any) to be chosen; there's a strong likelihood that this will also fail (needlessly duplicating a fruitless search) *and* will quite possibly close off backward paths to points from which an actual solution could be found.

   In fact the usual upshot (from observation) of trying a reversible square choice is that a handful of squares in mutually exclusive constraint zones end up dealing out

the same slightly larger handful of values in an apparently never-ending set of nested loops. This is an example of inappropriate use of CHOICE, since there is no division into "wrong" and "right" squares, and therefore nothing to be gained from trying another one on failure.

3. The choice of value, on the other hand, is clearly critical: probably only one will lead to a solution.; this should be the only reversible CHOICE in the program. If no valid number exists, then a previous assignment must have been wrong.

4. The assigned value is now no longer available to those blanks constrained by the current square, so it must be removed from all of their "possible" assignment sets (a subset of POSSIBLE itself).

5. This should always leave at least one remaining possible value for a square; an empty set here indicates we cannot find a solution given this assignment. This is the test for the reversibility guard, which will provoke backtracking.

6. The action guarded is updating the POSSIBLE set (described in more detail below).

4 Implementation

By virtue of the direct availability of set declarations and operations, many of the data model specifications and algorithm operations translate quite naturally into RVM-Forth, allowing for the postfix conversion. A simple example is the row generator *row()* — recall the function specification in (1) above on page 3, which becomes the following definition:

```
: GENROW ( n -- n.n.*.P )
   INT { , } XY PROD ;
```

The first line is the name and a comment with the stack effect; translating as "integer in, set of integer pairs out", as the specification says. `INT { }` is one way of creating an integer set, and here the comma between the curly braces allocates space for whatever is on top of the stack, within that set. This gives us `{n}` ; `XY` puts the set of column numbers on the stack while `PROD` generates the cartesian product of these two sets. The issue of garbage collection for such anonymous dynamically-created sets is addressed in appendix B. On its own, GENROW looks like this in action:

```
5 GENROW .SET
{(5,0),(5,1),(5,2),(5,3),(5,4),(5,5),(5,6),(5,7),(5,8)} ok
```

The union of this with the outputs of GENCOL and GENSECTOR form the set of squares in the constraint zone, returned by GENCZONE. With the square (3,2) on the stack (single line of output split for document):

```
DUP .PAIR CR GENCZONE .SET  (3,2)
{(0,2),(1,2),(2,2),(3,0),(3,1),(3,2),(3,3),
(3,4),(3,5),(3,6),(3,7),(3,8),(4,0),(4,1),
(4,2),(5,0),(5,1),(5,2),(6,2),(7,2),(8,2)}ok
```

Part of the initialisation involves the set POSSIBLE, wherein each blank square is paired with its available values: generation of the latter is performed by AVAILABLE. As described above in section 2.1, what is required is the set of digits in DIGIT which are *not*

in the already assigned squares within the constraint zone of the square (r, c) in question; this is specifiable in set notation as:

$$DIGIT - board(\!|\ czone(r, c)\ |\!)$$

Converted for postfix and stack adjustment, it maps exactly to RVM-Forth:

```
: AVAILABLE ( n.n.* -- n.P )
   GENCZONE BOARD SWAP IMAGE DIGIT SWAP \ ;
```

Using the same square as the last example, with a sample puzzle loaded:

```
DUP .PAIR CR AVAILABLE .SET  (3,2)
{1,2,5,8,9}ok
```

4.1 The Update procedure

A more detailed look at this is now presented, it being the section where the reversibility guard comes into play. A boolean variable called VALID acts as a flag for the guard, initialised to *true*. Arguments are the square and the value just assigned to it. Outline:

1. Generate the subset of POSSIBLE which needs updating. This is simply done by finding the constraint zone of the square with GENZONE, and performing a domain restriction (the word `<|` in the definition below: described in appendix A) on POSSIBLE. If the resulting set is not empty, the loop now builds a new set of this type.

2. (Begin loop) For each blank square in this set, remove the assigned value from its set of available values, where present. If the resulting set is empty, set VALID to *false* and exit loop. Otherwise, add the new pair (square, remaining possible values) to the set being built.

3. (End loop) Test VALID: if false, instigate reversal to previous CHOICE (and make another choice). Otherwise, use function override (`<+`) to update POSSIBLE with the new sets of values for the affected squares.

The RVM-Forth code to achieve this is shown below. ASSIGNED is a variable holding the last assigned value, and UNPAIR leaves the first and second elements in that order on the stack.

```
1.  : UPDATE-POSSIBLE ( n.n.*  n -- )
2.     to ASSIGNED GENCZONE POSSIBLE <|
3.     DUP ?{} NOT IF                      ( test for non-empty set )
4.        INT INT PROD INT POW PROD {
5.           DUP CARD 0 DO
6.              DUP I @ELEMENT            ( index through elements )
7.              UNPAIR ASSIGNED SUBTRACT-ELEMENT
8.              DUP ?{} IF FALSE to VALID LEAVE ELSE |->P,S , THEN
9.           LOOP } VALID --> POSSIBLE SWAP <+ to POSSIBLE  THEN DROP ;
```

Line 4 This is the RVM postfix way of specifying a pair comprising a pair of integers and a set of integers (i.e. a square plus set of values)

$$((r, c), \{v_1, v_2, \ldots, v_n\})$$

Line 8 Having subtracted the assigned value, we now check the remaining set with the empty set test. In theory, the guard might go directly after this, but reversing from the middle of a set-building operation is incompatible with the reversibility mechanism; a boolean is used so the guard can be outside.

Line 9 The symbol for the guard is `-->`, here testing VALID. The symbol is cognate with the General Substitution Language's $\Longrightarrow$, normally associated with an IF...THEN construct in a B specification[2]. Here and in the proposed RB0 language, it functions as a "naked" guard, prompting backtracking on failure.

5 Performance

In terms of raw speed, running in a virtual machine atop a subsystem for sets (albeit an efficient one) is unlikely to be optimal. Instead we balance the simplicity of the development using a naïve heuristic against the number of tries the program needs to solve a problem designated "very hard", or "fiendish" (anything less challenging requires little if any backtracking).

While the choice of square could simply take the first one encountered (using ELEMENT), there is a non-backtracking random choice available called PCHOICE, and along with a reversible RANDOM-CHOICE for the value assignment, this gives a variety of possible paths for the program to follow. Sometimes a solution will be found very quickly even for the hardest puzzles in this way. The average range for puzzles encountered so far is 150-500 attempted assignments. The unconstrained search space for assigning about 50 squares from 9 potential values for each one is not considered further here.

6 Further Work

The initial approach here has been to attack both ends of the problem first to gain an idea of what a complete development should encompass. While refinement from the abstract machines will often use the usual techniques, certain differences will be apparent: in the data model, sets need not be refined away as they are directly implementable; also the reversible computations introduce certain differences in refinement methods (and associated proofs), outlined in [6]. Issues thrown up by refinement of this case study will therefore require investigation.

6.1 Code generation: stack vs locals?

The code generation stage is in very early infancy as yet; translation schemas to support this have been begun, but certain questions arise. Forth, RVM or otherwise, is a stack-based language, and much of its operational simplicity derives from not having to declare and handle local variables for basic operations. B, on the other hand, is in the tradition of variable-manipulation languages, and this provides an uncomfortable meeting-point for the two modes.

Provision of an explicit stack at the specification level would cause more problems than it would solve, leaving two alternatives. Ideally, we would aim for a transparent "under the hood" translation system, whereby appropriate use was made of the Forth stack from a standard local-using specification. Initially, however, translation will (relatively) simply

[2]Such programming-style constructs are "syntactic sugar" for the underlying GSL notation.

map B locals to RVM locals, less than optimal though this will prove; meanwhile a reliable way of optimising the translation must be investigated.

The combination of top-down approach from specification and bottom-up approach from code will provide illumination from two aspects for the development of a middle stage, hypothetical as yet, an *implementation-level* specification language closely based on the existing B0.

References

[1] Jean-Raymond Abrial. *The B Book*. Cambridge University Press, 1996.

[2] W J Stoddart. Efficient reversibility with Guards and Choice. In M A Ertl, editor, *18th EuroForth*, 2002. Available from: http://www.complang.tuwien.ac.at/anton/euroforth2002/papers/bill.rev.ps.gz.

[3] W J Stoddart. Using Forth in an Investigation into Reversible Computation. In P Knaggs and M A Ertl, editors, *19th EuroForth*, 2003.

[4] W J Stoddart. RVM-Forth, a Reversible Virtual Machine: User Manual. In M A Ertl, editor, *19th EuroForth*, 2004. Available from http://dec.bournemouth.ac.uk/forth/euro/ef04/stoddart04.pdf or http://www.scm.tees.ac.uk/formalmethods/index.php.

[5] W J Stoddart and F Zeyda. Implementing sets for reversible computation. In A ERTL, editor, *18th Euroforth, Technical University of Vienna*. 2002.

[6] F Zeyda, W J Stoddart, and S E Dunne. The Refinement of Reversible Computations. In T Muntean and K Sere, editors, *2nd International Workshop on Refinement of Critical Systems*, 2003. Available from www.esil.univ-mrs.fr/ spc/rcs03/rcs03.

A Set operations

Many set operations are provided "out of the box" with RVM-Forth, and more advanced ones can be built up with little trouble. For instance, from two sets (of arbitrary types), the set of all possible pairs (cartesian product) can be generated with PROD:

```
SAVOURY SWEET .SET .SET
{chocolate,fruit,honey} {cheese,fish,onion} ok
SWEET SAVOURY PROD .SET
{(chocolate,cheese),(chocolate,fish),(chocolate,onion),
(fruit,cheese),(fruit,fish),(fruit,onion),
(honey,cheese),(honey,fish),(honey,onion)} ok
```

Tasty. As usual, the order of the stack parameters is the same as for the infix operator: $S \times T$ becomes `S T PROD`.

Working with relations and functions is also straightforward. We define a simplistic telephone directory (a relation from strings to integers) to work with, called PHONE, which looks like this:

```
CR PHONE .SET
{(Bill,2673),(Frank,4611),(Keerthi,4611),(Michelle,4611),
(Rob,4611),(Steve,2657)}
```

An operation used in the text is domain restriction (the word `<|`), in which a set of values from the domain is used to extract only those pairs which have a first value in that set — the result being returned as another set. For instance, with a query set dynamically constructed:

```
STRING { " Michelle" , " Keerthi" , } PHONE <|  ok.
.SET {(Keerthi,4611),(Michelle,4611)}ok
```

A relational image is really just the range of this result, though it's normally implemented as a separate operation. The set notation $S(\!|\ U\ |\!)$, where U is a subset of the domain of S, becomes `S U IMAGE` in RVM-Forth:

```
PHONE STRING { " Michelle" , " Keerthi" , } IMAGE ok.
.SET  {4611}ok
```

Which is the range of the previous result. Range restriction is the mirror image of domain restriction (the word `|>`); notice the stack parameters are the other way round to reflect the ordering of the infix operator:

```
PHONE INT { 4611 , } |>  ok.
.SET  {(Frank,4611),(Keerthi,4611),(Michelle,4611),(Rob,4611)}ok
```

A certain overcrowding is becoming apparent. Some rehousing later, the outdated phone numbers can be overwritten with function override. The word `<+` is defined as:

```
: <+ ( s1:x.P s2:x.P -- s3:x.P  where s3 = s1 <+ s2 )
DUP DOM ROT <<| \/ ;
```

Using `<<|`, which is domain subtraction (the complement of domain restriction), this removes the pairs from s1 which are due to be updated, then unions the remainder with s2. To update our phones, we construct a set UPDATES:

```
{(Frank,2680),(Rob,3719)}
```

and invoke function override to effectively overwrite those pairs whose first value matches one of the first values in the update set. Thus:

```
PHONE UPDATES <+ to PHONE ok
CR PHONE .SET
{(Bill,2673),(Frank,2680),(Keerthi,4611),(Michelle,4611),
(Rob,3719),(Steve,2657)}ok
```

This is of course used in the SuDoku program to update the POSSIBLE set with reduced sets of values for the affected squares only.

B Garbage Collection

As might be imagined such wanton creation of arbitrary sets has huge potential for garbage creation; this is automatically collected during reverse execution, but not necessarily otherwise. However, the potential-value capabilities of the RVM allow for the provision of a wrapper which ensures garbage collection for this sort of program. The details are covered in [4], but the results of using these capabilities are shown here with the aid of a diagnostic tool called Heapwatch (© Frank Zeyda). On starting the RVM itself:

```
HW-STATS HeapWatch: Statistical Information:
Current memory in use: 936 bytes (0 KB)
Number of calls to malloc(): 40
Number of calls to calloc(): 0
Number of calls to realloc(): 33 (24 ret. same + 9 diff. address)
Largest memory allocation: 56 bytes in file setkernel.c, line 252.
Average allocation size: 26 bytes
Peak memory utilisation: 936 bytes (0 KB) + 2625 KB for HeapWatch.
```

After a single run of the solver without garbage collection invoked, the situation is as below — followed immediately by another run and the memory report.

```
HW-STATS HeapWatch: Statistical Information:
Current memory in use: 211816 bytes (206 KB)
Number of calls to malloc(): 38819
Number of calls to calloc(): 0
Number of calls to realloc(): 30506 (19868 ret. same + 10638 diff. address)
Largest memory allocation: 344 bytes in file setkernel.c, line 211.
Average allocation size: 44 bytes
Peak memory utilisation: 211816 bytes (206 KB) + 2625 KB for HeapWatch.

    ( ******** another puzzle solved here ********** )

HW-STATS HeapWatch: Statistical Information:
Current memory in use: 427324 bytes (417 KB)
Number of calls to malloc(): 56596
Number of calls to calloc(): 0
Number of calls to realloc(): 43834 (29158 ret. same + 14676 diff. address)
Largest memory allocation: 344 bytes in file setkernel.c, line 211.
Average allocation size: 44 bytes
Peak memory utilisation: 427324 bytes (417 KB) + 2625 KB for HeapWatch.
ok..
```

Clearly some garbage is being left behind, and would continue to build up. Using the `<TRY S CUT>` construct to wrap the program, however, allows it to run, print (or store) its output, and then garbage associated with the run is collected. From a similar cold start to the first quoted above, a run now leaves the system in this situation:

```
HW-STATS HeapWatch: Statistical Information:
Current memory in use: 936 bytes (0 KB)
Number of calls to malloc(): 38819
Number of calls to calloc(): 0
Number of calls to realloc(): 30506 (19866 ret. same + 10640 diff. address)
Largest memory allocation: 344 bytes in file setkernel.c, line 211.
Average allocation size: 44 bytes
Peak memory utilisation: 211816 bytes (206 KB) + 2625 KB for HeapWatch.
```

So the memory allocated by the frequent calls to `malloc()` or `realloc()` has now all been reclaimed.

C RVM-Forth Implementation Code

```
( ========== Declarations and initial Board ================== )

1 9 .. VALUE DIGIT
0 8 .. VALUE XY
NULL VALUE_ BOARD        ( Reversible variable )
NULL VALUE_ POSSIBLE     ( Reversible variable )
1 VALUE LOOPS 1 VALUE TRIES ( Bookkeeping )
8 VALUE COLI 8 VALUE ROWI 0 VALUE ASSIGNED
TRUE VALUE_ VALID ( Set to false if square left with no domain )

( Generalised board-builder; assumes 81 numbers loaded on stack)
: BUILD-BOARD ( n TIMES 81 --  )
   8 to ROWI
   INT INT PROD INT PROD {
      BEGIN ROWI -1 >
      WHILE
         8 to COLI
         BEGIN COLI -1 >
         WHILE
            DUP 0= NOT
            IF ROWI COLI |->I,I SWAP |->P,I ,
            ELSE DROP
            THEN
            COLI 1- to COLI
         REPEAT
         ROWI 1- to ROWI
      REPEAT } to BOARD ;

( ================ Utilties; zone generation etc ================)

: UNPAIR ( x1.x2.* -- x1 x2 )
   DUP FIRST SWAP SECOND ;

: GENROW ( n -- n.n.*.P )
   INT { , } XY PROD ;

: GENCOL ( n -- n.n.*.P )
   XY SWAP INT { , } PROD ;

( Here, nr is the row number, and nc isn't.
  The output is the row & col of the sector index )
: SECTORINDEX ( nr nc -- n n )
   DUP 3 MOD - SWAP
   DUP 3 MOD - SWAP ;

: GENSECTOR ( nr nc -- n.n.*.P )
   SECTORINDEX
   DUP 2 + .. SWAP
   DUP 2 + .. SWAP PROD ;

( Find the constraint zone for a particular square )
: GENCZONE ( n.n.* -- n.n.*.P )
   UNPAIR DUP
```

```
   GENCOL  ROT ROT OVER
   GENROW  ROT ROT
   GENSECTOR  \/ \/ ;

( Now we find the values a blank square can take )
: AVAILABLE ( n.n.* -- n.P )
   GENCZONE BOARD SWAP IMAGE DIGIT SWAP \ ;

( ============================================================ )
( Pretty(ish)-print subsystem. Can safely be ignored )

0 VALUE ELEMINDEX NULL VALUE BOARDSIZE

: VLINE 124 EMIT ;
: LINE VLINE CR ." +---------+---------+---------+" CR  ;

( Convert co-ord pairs to scalar square numbers)
: CONVERTBOARD ( n.n.*.n.P -- )
   INT INT PROD {
      DUP CARD 0 DO
         DUP I @ELEMENT
         UNPAIR SWAP UNPAIR SWAP 9 * +
         SWAP |->I,I ,
      LOOP
   } NIP ;

: .BOARD ( -- )
   CONVERTBOARD
   0 to ELEMINDEX DUP CARD to BOARDSIZE
   81 0 DO ELEMINDEX BOARDSIZE < IF
         DUP ELEMINDEX @ELEMENT
      ELSE DUP 0 @ELEMENT
      THEN
      I 27 MOD 0=
      IF  LINE VLINE ELSE I 9 MOD 0=
         IF  VLINE CR VLINE ELSE I 3 MOD 0=
            IF VLINE
            THEN
         THEN
      THEN
      DUP FIRST I = IF
         SPACE SECOND . ELEMINDEX 1+ to ELEMINDEX
      ELSE 3 SPACES DROP THEN
   LOOP LINE DROP CR ;
( ==================== End of print system ================== )

( Build set of squares and their possible values )
: INIT-POSSIBLE ( -- )
   XY XY PROD BOARD DOM \
   INT INT PROD INT POW PROD {
      DUP CARD 0 DO
         DUP I @ELEMENT
         DUP AVAILABLE |->P,S ,
      LOOP
    } to POSSIBLE DROP ;
```

```
( Get the next squares from the most constrained -- returns a
  subset of POSSIBLE. We cheat a bit by using the set ordering
  to find the lowest card in the range of POSSIBLE )
: GETSQUARES ( n -- n.n.*.n.P.*.P)
   0 (: VALUE N :)
   POSSIBLE DUP RAN ELEMENT CARD to N
   INT INT PROD INT POW PROD {
      DUP CARD 0 DO
         DUP I @ELEMENT
         DUP SECOND CARD N =
         IF , ELSE DROP THEN
      LOOP
   } 1LEAVE ;

( Picks next square and its availables (an element of POSSIBLE)
  to send to assign; subtract it from POSSIBLE )
: NEXTUP ( n.n.*.n.P.*.P -- n.n.*.n.P.* )
   PCHOICE DUP POSSIBLE SWAP
   SUBTRACT-ELEMENT to POSSIBLE ;

( Assigns from square and set of values a single value, adding
  pair to Board; leaves square and value separately )
: ASSIGN ( n.n.*.n.P.* -- n.n.*  n )
   UNPAIR RANDOM-CHOICE TRIES 1+ to TRIES
   2DUP |->P,I BOARD SWAP ADD-ELEMENT to BOARD ;

( Remove assigned from the domain of each square in the constraint
  zone of the last assigned square -- update by func override. Should
  not attempt to update when no blanks need updating )
: UPDATE-POSSIBLE ( n.n.*  n -- )
   to ASSIGNED GENCZONE POSSIBLE <|
   DUP ?{} NOT IF
      INT INT PROD INT POW PROD {
         DUP CARD 0 DO
            DUP I @ELEMENT
            UNPAIR ASSIGNED SUBTRACT-ELEMENT
            DUP ?{} IF FALSE to VALID LEAVE ELSE |->P,S , THEN
         LOOP } VALID --> POSSIBLE SWAP <+ to POSSIBLE  THEN DROP ;

( ============ Run and step-through facilities ================ )
: START  ( -- )
   NULL to BOARD NULL to POSSIBLE
   1 to TRIES 1 to LOOPS
   32 WORD LOAD-FILE BUILD-BOARD INIT-POSSIBLE ;

: STEP  ( -- )
    GETSQUARES NEXTUP
    ASSIGN UPDATE-POSSIBLE LOOPS 1+ to LOOPS ;

: SOLVE ( -- n.n.*.n.*.P )
   BEGIN
      BOARD CARD 81 <
   WHILE
      STEP
   REPEAT  ;
```

```
( Takes filename after word, e.g.: TRY-TO-SOLVE S1 )
: TRY-TO-SOLVE ( -- ) CR
   START SOLVE BOARD .BOARD  TRIES . ." TRIES IN "
   LOOPS . ." LOOPS. " CR ;

( Garbage collecting wrapper for above )
.( TRY <filename> runs the puzzle )
: TRY
   <CHOICE
       <TRY TRY-TO-SOLVE CUT>
     []
       CR
   CHOICE> ;
```

First experiences with Microcore

N.J. Nelson, C. Williams

Abstract

Following the convincing demonstrations at EuroForth 2004, we decided to use the "Microcore" VHDL Forth processor in the design of three new products. This paper will describe our progress in expanding the core design with additional peripherals, performing simulation, board implementation, and early experiments in writing code on the Microcore.

N.J. Nelson B.Sc., C.Eng., M.I.E.E.
Micross Electronics Ltd.,
Units 4-5, Great Western Court,
Ross-on-Wye, Herefordshire.
HR9 7XP U.K.
Tel. +44 1989 768080
Fax. +44 1989 768163
Email. njn@micross.co.uk

C. Williams B.Sc., C.Eng., M.I.E.E.
Chrysalis Design,
Craig-y-don,
Llandinam,
Powys
SY17 5BG
Tel. / Fax. +44 1686 688065
Email. chris@chrydesn.demon.co.uk

1. Overview of Microcore

Microcore is a VHDL description of a microcontroller, which can be implemented in an FPGA. It is highly configurable, and in particular, the external data path width is a compilation variable, so that various different "sizes" of the same processor may be constructed, with different performance / cost balances. The code for Microcore is available under a licence which is similar to open source software, and which encourages other to contribute to the project development while retaining compatibility and openness.

Microcore was first described by Klaus Schleisiek at the 17th EuroForth at Dagstuhl, and he also described an implementation of the device at the 20th conference, where he gave a convincing demonstration of the technology.

Microcore has its own website from which the code may be downloaded.

2. Reasons for choosing Microcore

Advantages unique to Microcore

a) It's free. This is a serious consideration for a small company where development budgets are tight.
b) It comes from a known and trusted developer. Klaus also designed the IX1 microcontroller which has given us years of completely trouble-free service.
c) You can actually talk to the designer, who even answers email and telephone calls. This is in marked contrast to other offerings of standard cores.
d) Genuine futureproofing
Even if the hardware goes obsolete, the software won't. There should be no difficulty in moving a Microcore project to a future FPGA technology. The struggle to buy "one careful previous owner" RTX chips will be over.
e) Control
We have all the code to produce versions of Microcore for as long as we need to.
f) No black boxes
If there is a problem, nothing is hidden. We can analyse the problem to whatever depth is required.
g) Simple and inexpensive design tools
We have used Xilinx and Mentor Graphics tools.
h) Different sizes, same code
We can use exactly the same software on both 8 bit and 32 bit external data bus width versions.
j) Simplicity
We almost understand quite a bit of it.
k) It runs Forth
All of us understand it, and with careful core design it should be possible to port large chunks of our existing code straight in.

Advantages of FPGA microcontrollers over fixed hardware

l) Potential for future performance enhancement
As the speed of FPGAs increases, so will the speed of Microcore.

m) Extensibility
On-chip peripherals can be added relatively easily.
n) Pinout flexibility
Pinouts can be matched to the PCB layout requirements, enabling a simpler and less expensive 4 layer PCB to be used. Without this, a 6 layer PCB would almost certainly be needed.

3. Our particular requirements

We needed to replace and upgrade three products.
a) The Virtual Programmable Logic Controller
This is a high integrity device which provides the central control functions of a distributed automation system. We described this card at EuroForth 97. It uses the RTX2001 as its CPU, and has a PCI interface with a PC but is otherwise completely autonomous. This has been a very satisfactory design with excellent reliability.
b) The Rapid Automated Bacterial Impedance Technique (RABIT), also a PCI card but this time designed as a centralised data logger for a large number of distributed microbiological tests cells.
c) The RABIT block module, which provides ultra-accurate temperature control and digitisation of a group of 32 microbiological test cells.

Both RABIT circuits used the Intel 251 microcontroller, which is possibly the worst microcontroller ever produced. We shall be very glad to replace it.
The new versions of both a) and b) are very similar, using 32 bit data bus widths and an Ethernet connection to the PC instead of a PCI connection. They have differing power supply, communication and memory requirements.
The new version of c) will be an 8 bit implementation.

4. Experiences with the tools

Design philosophy

Our basic design philosophy is to make it simple and to use as much of the Microcore design as possible. We don't want to have to dig deep into the VHDL to understand it and by doing a conservative design where we keep a respectful distance from the limits we hope to reduce our problems. We also need to remember that the number of boards that we will make is quite small, and that the cost of the development tools must be kept to a minimum.

The chips used in the design also affect the tools. Each chip vendor has its own tool set for which it is optimised, but limited to its own ICs, this includes both Lattice and Xilinx . By choosing a class of chip that has already been used to implement Microcore other potential pitfalls may be reduced. Microcore has already been implemented in the Xilinx Spartan series of chips. These are currently cheaper than the Lattice parts, but they do require an external flash memory to initialise them. We want to reduce manufacturing problems so we don't want ball grid array packages. We also want a part that will give room for experimentation in the future.

The XC3S400 PQ208 is a Xilinx Spartan 3 device in a 208 pin plastic quad flat pack and it will accept the Microcore with room for expansion and the additional peripherals that we need. A cheap programmer is available for transferring the compiled output on the computer to the flash memory on the board and modifying the design as often as required.

Choice of tools

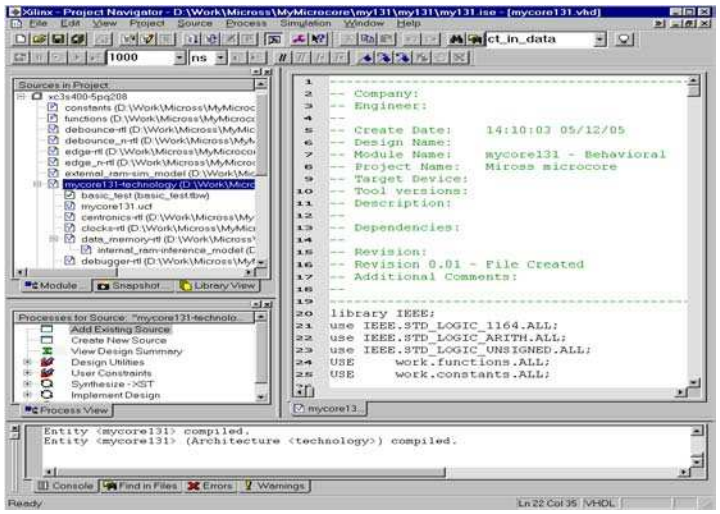
There are two possible tool sets we could use, the Xilinx ISE (Integrated Software Environment), or the Synplify system from Synplicity. The pros and cons of each are as follows:

Xilinx ISE	Synplify
Free	~£10,000
Complete design from beginning to end	Works with Xilinx tools
Limited optimisation increases chip area used.	Advanced optimisation gives smallest possible design
Design may not give maximum possible speed.	Advanced optimisation may give fastest design

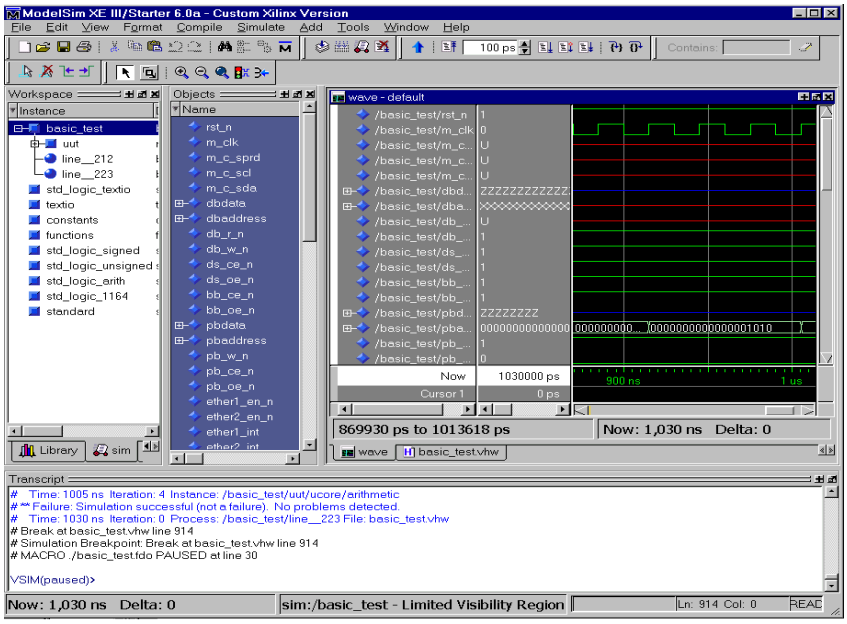
The ModelSim simulator from Mentor Graphics is provided to simulate the designs at all levels. This accepts a VHDL description that can be functional, i.e. no timing information, and allows the VHDL to be checked for accuracy, right up to a full post layout description that gives detailed operations and timings.

We chose to use the Xilinx ISE foundation pack that can be downloaded free from the Xilinx web site. By not pushing the design to its limits we hope that the reduced optimisation will not cause a problem. The huge reduction in cost is also more in line with the number of chips we are likely to produce.

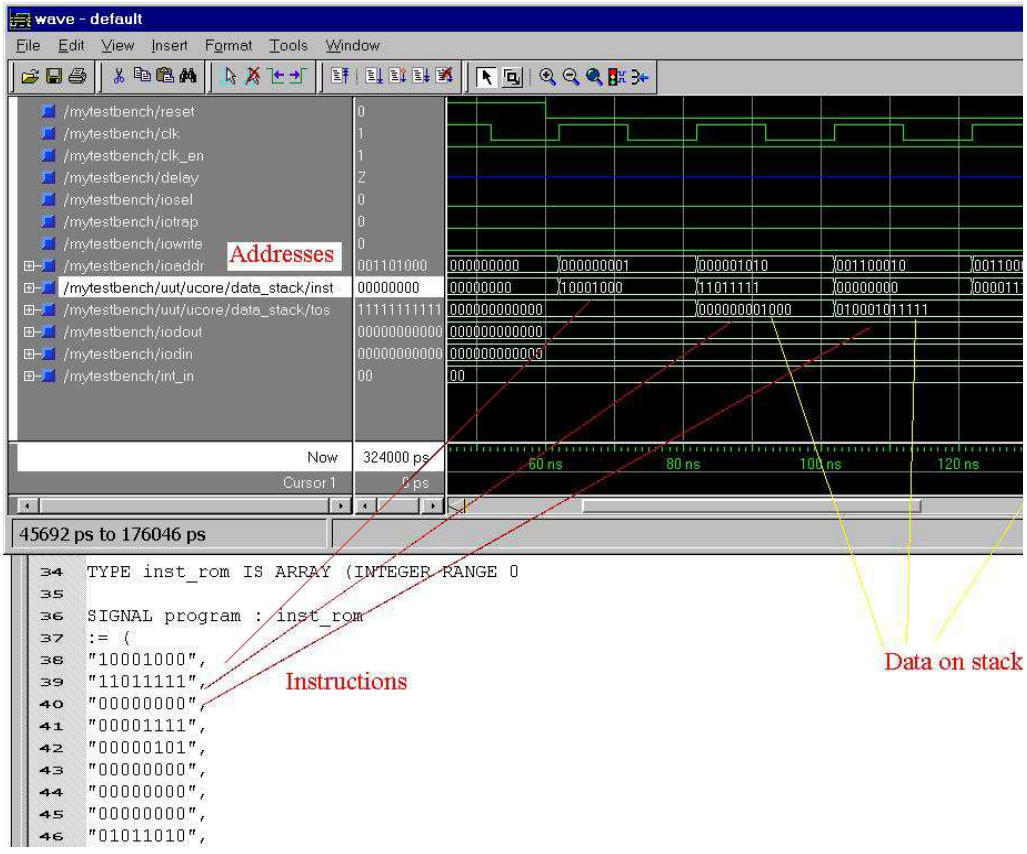
Here is a typical screen for the ISE version 7:
This shows a project window where all the files can be entered, a process window where for each file all the possible actions are listed, an edit window where all the files can be viewed and edited and a console where progress and errors are displayed.



From here, all the different operations needed to build a design are managed. One obvious operation is to run the simulator. By selecting the test bench file, which contains waveforms, you have the option to run the simulator directly. A typical screen shot:



This again shows a multi-window screen with the waveform result on the right.
The first job in evaluating the tools and the microcore design was to try and run a functional simulation.



This shows the first instructions in the boot memory being run at the end of reset and loading immediate data onto the stack. The first 2 instructions have the top bit set that then loads the following 7 bits onto the stack.

5. Peripherals we have developed

At this stage of using Microcore we need two additional peripherals:

- Watchdog counter
- SPI serial interface

Watchdog counter.

The watchdog counter counts a period of time, using the master input clock as its reference and if it is not reset in that period by the software it causes a processor reset. We require a timeout of 1ms and can set this directly into the hardware based on the processor master clock.

The VHDL code is as follows:

```
watchdog_reload <= '1' when sel_io = '1' AND
(addr(watchdog_select_address_bit) = '1') else '0';

watchdog_control : process(m_clk,rst_n,watchdog_reload,watchdog_div)
BEGIN
      if(rst_n = '0') then
            --On reset set a slightly longer watchdog time
            watchdog_div <= (OTHERS => '1');
      else
            if (rising_edge(m_clk)) then
                  if watchdog_reload = '1' then
                        watchdog_div <= "11000011010100000000"; --200000
                  else if two_meg_div = "0000" then
                              watchdog_div <= watchdog_div - 1;
                        end if;
                  end if;
            end if;
      end if;

END PROCESS watchdog_control;
```

This describes a simple down counter watchdog_div that is decremented on every rising edge clock with the code:

```
      watchdog_div <= watchdog_div - 1;
```

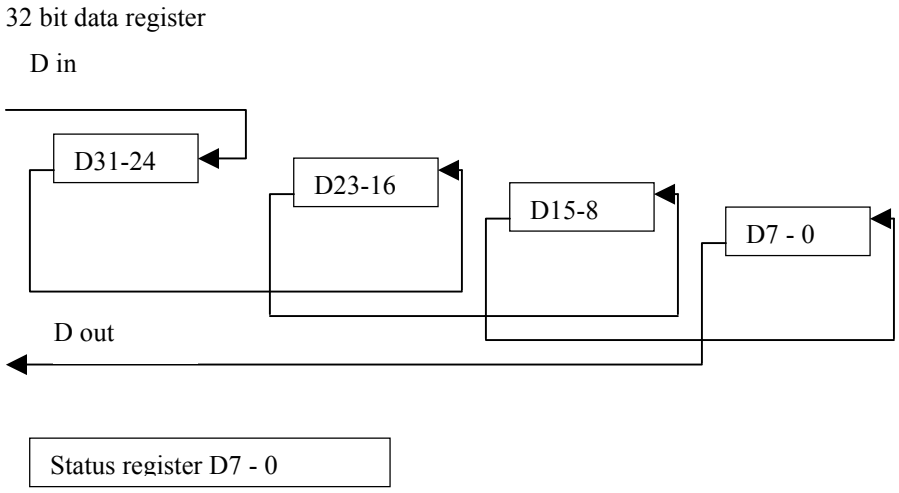
This is modified if we have a reset signal or a watchdog_reload signal. The watchdog reload signal comes from a memory access instruction from the processor to the watchdog address.

The microcore reset generator then looks at both the external reset signal and the value of watchdog_div. If watchdog_div ever gets to a value of ‘0’, the processor is reset, and can start again.

SPI serial interface

The SPI (Serial Peripheral Interface) is more complex than this. It uses a clock line, two data lines (one input and one output) and a chip enable to provide bidirectional data transfer, and can be used to talk to a wide range of chips. We currently need to communicate with a serial flash memory to store our programs. Normally, data transfers are in 8 bit bytes, but we have made good use of the 32 bit data path to allow up to 4 bytes to be transferred at a time without processor intervention.

The general hardware arrangement is as follows:



Bit	Function	Status register	
0	Int	Read, cleared when written	
1	Start	Write, cleared when done	
2	B0	Write	Byte count to transfer
3	B1	Write	
4	CE1	Write	Chip enable 1
5	CE2	Write	Chip enable 2

The 32 bit shift register sends data out from the low order byte, and reads data in through the high order byte. The status register, a memory location in the I/O memory area holds 5 bits to control the operation:

- An interrupt bit to indicate when a transfer is complete.
- A start bit, set by the user to start a transfer and cleared automatically when the transfer is complete.
- A two bit count of the number of bytes to be transferred, set by the processor.
- A pair of chip enables, passed directly to the devices, set by the processor.

The VHDL for this has been developed as a separate module. This contains the status register, all the shift registers and counters. It is controlled by a hardware state machine implemented as follows:

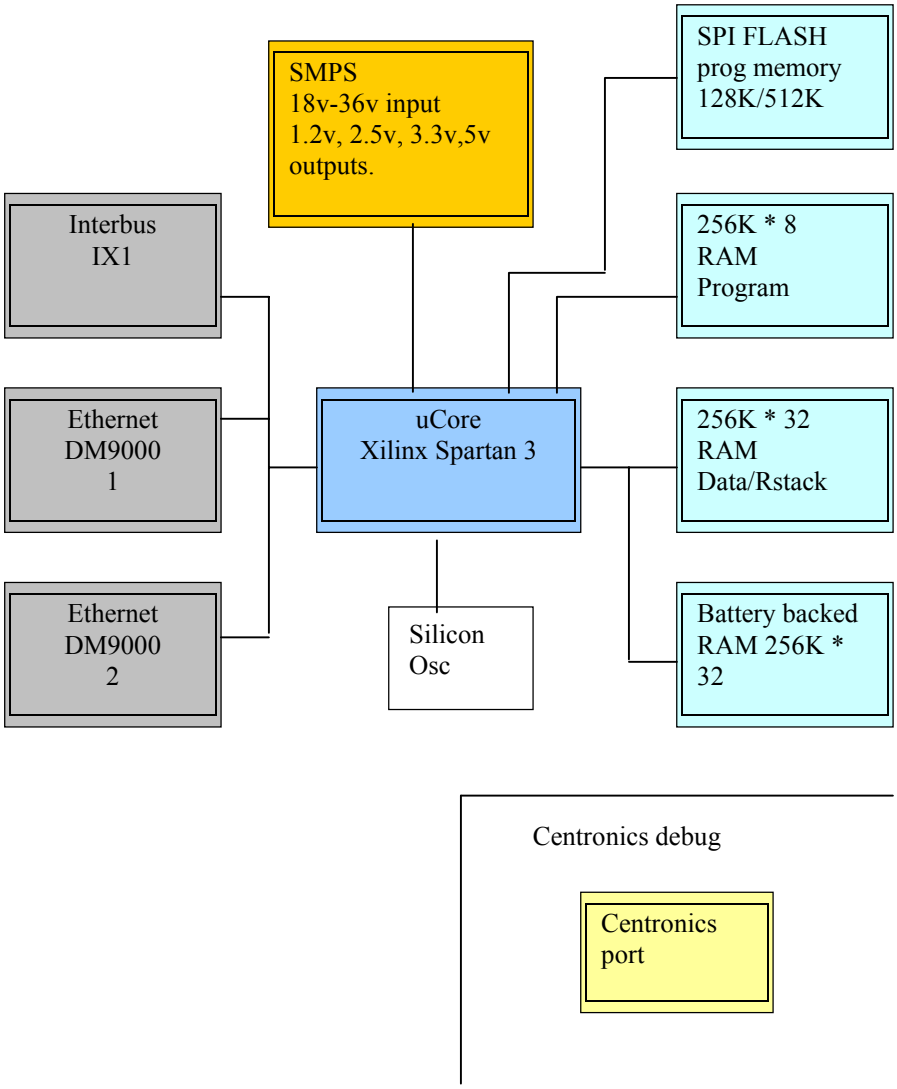
```
State_machine: process(reset,state,status_reg,shift_clock)
BEGIN
   if reset = '1' then
            state <= waiting;
            bit_counter <= "000000";
       elsif falling_edge(shift_clock) then
            case state is
                  when waiting => if status_reg(start_bit) = '1' then
                        state <= running;
                        if status_reg(byte_count_1_bit downto
byte_count_0_bit) = "00" then
                              bit_counter <= "000111";
                        elsif status_reg(byte_count_1_bit downto
byte_count_0_bit) = "01" then
                              bit_counter <= "001111";
                        elsif status_reg(byte_count_1_bit downto
byte_count_0_bit) = "10" then
                              bit_counter <= "010111";
                        else
                              bit_counter <= "011111";
                        end if;
                  end if;
                  when running => bit_counter <= bit_counter - 1;
                        if bit_counter = 0 then
                              state <= waiting;
                        end if;
                  when others => state <= waiting;
            end case;
       end if;
END PROCESS state_machine;
```

As you can see, the naming conventions and appearance are much closer to ‘C’ than to Forth, but you can also see that it is using its current ‘state’ which can be ‘waiting’ or ‘running’ along with the start_bit in the status register and the bit counter to control its operation.

In use you load the data to be transferred into the shift registers, set the start_bit, the byte count and the chip enable, and wait either for the interrupt or by poling the start bit for the end of the transfer. Any data read back from the device can then be read into the program from the shift registers.

6. The first hardware design

This is the block diagram of the complete system that we are building:



The most important question to ask at the start of the design is at what voltage the chips will run. The Xilinx chip requires 1.2V and 2.5V for its internal operation but will interface to the outside world at any voltage up to 3.3V. Looking at the chips around it, some are available at 3V, some at 3.3V and others at 5V.

In this case most chips are available for 3.3V operation, except for the IX1 chip that is only available at 5V. This meant providing a separate 5V supply using voltage converters on the signals to and from the Microcore.

The next question to ask is how the memory is to be organised. We need 32 bit wide RAM for stack and data, but we also need an area of battery backed ram for long term storage. The memory needs to be 10ns to run at full speed and we could not find memory that fast that was also low power enough to be battery backed. Our solution is to have both kinds of memory, with the 55ns battery backed ram requiring 2 cycles for access.

The program memory is only 8 bits wide but requires the same compromises. It needs to be fast RAM and non-volatile. We could not find anything to do this. We compromised with a fast RAM chip and a slow serial flash memory. The Microcore can be made to write to its program memory, so at boot time, running the internal boot loader, the code in the flash memory can be read out and written to the RAM. The program then jumps to the start of the RAM. The flash memory can be written by the program as well.

The external peripherals that we need for the application are placed on the memory bus. The Centronics debug port comes straight from the Microcore design and is used in initial development for programme load and debugging. The master clock is a silicon oscillator, which is an alternative to a crystal. This has the advantage both of size and its ability to 'jitter' slightly. This does not affect the operation of the Microcore, but it does reduce the electromagnetic interference and that helps with technical approvals.

The design was started using version 1.30 of the Microcore. Part way through the process 1.31 became available. This has some significant differences and required some changes to our designs. Then 1.32 became available. The rapid changes can cause problems in the design. It is better to stay with a version until its limitations cause real problems rather than change every time a new version is available.

7. First steps in software development

We expect to have some hardware to show in time for the conference. With luck, a little software might even have been written.

8. Conclusions

We'll tell you next year!

Self Documenting Sequences

N.J. Nelson, K.B. Swiatlowski

Abstract

When creating automation code for mechanical handling equipment which is specially adapted for installation at a wide variety of different sites, it is common for numerous alterations in the code to be required on site, at the last minute. Under pressure, user documentation starts to diverge from code. How nice it would be, if clear, accurate and readable documentation could be regenerated automatically each time the code was recompiled - and translated, also automatically, into the customer's language!

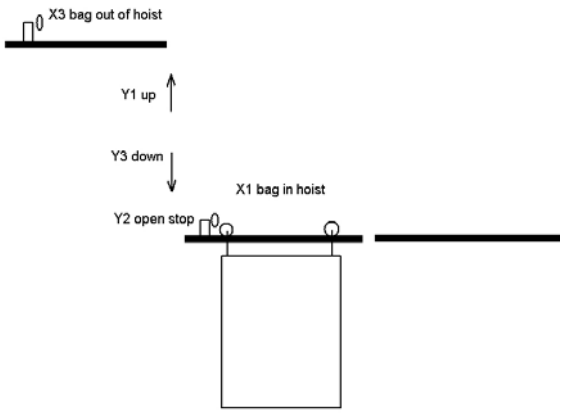
N.J. Nelson B.Sc., C.Eng., M.I.E.E.
K.B. Swiatlowski Mgr.
Micross Electronics Ltd.,
Units 4-5, Great Western Court,
Ross-on-Wye, Herefordshire.
HR9 7XP U.K.
Tel. +44 1989 768080
Fax. +44 1989 768163
Email. njn@micross.co.uk

1. Background - sequences

At the 1997 EuroForth conference, Jonathan Morrish won a prize for his paper, "*Rapid development of real time multi-sequence control programmes* ".
Jonathan is no longer with Micross, but the wordset he described was so successful that we're still using them today.

In a typical complex conveyor system, the overall control process is broken down into a set of "sequences", each of which controls the movement of an item from one position to another. The sequences is divided into a series of "steps" thus forming a kind of state machine.

A simple example Jonathan used looked like:



A container runs by gravity into a vertical conveyor, is elevated, and runs off to a storage position.

The sequence would be described in the documentation as follows:

Sequence 1 - Hoist

Step	Stepped on by	Outputs
	Bag runs into hoist	
0	Wait for container to arrive in hoist (X1)None Hoist at bottom (X4)	
	Bag settles in hoist	
1	Wait for 3s for container to settle	None
	Hoist goes up	
2	Hoist at top (X2)	Hoist UP (Y1)
	Bag runs out of hoist	
3	Container out of hoist (X3)	Open stop (Y2)
	Hoist goes down	
4	Hoist at bottom (X4)	Hoist DOWN (Y3)

Since 75% of our equipment is exported, the documentation is generally provided in at least two languages.
In Jonathan's wordset, the sequence is coded as follows:

```
: HOIST
  1 SS CASE                       \ SS - set up specified sequence
    0 OF 1 X 4 X AND ?NS ENDOF    \ X - true if input on
    1 OF 3 SECS ?NS      ENDOF    \ SECS - true if time
    2 OF 2 X ?NS         ENDOF    \ ?NS - next step if true
    3 OF 3 X 20DNS       ENDOF    \ 20DNS - set diagnostic after 20s
    4 OF 4 X ?0S         ENDOF    \ 0S - reset sequence
  ENDCASE
  S@                              \ S@ - returns step number
    DUP     2 =   1 Y             \ Y - output on/off
    DUP     3 =   2 Y
            4 =   3 Y
;
```

Note how compact the code is.

2. The problem

a) All conveyor systems are different, so sequences need to be individually documented and coded for each installation.

b) Complete conveyor systems are very large and complex and are only completely assembled for the first time at the end customer's site.

c) At the time of installation it is always found that there differences between the original specification and the customer's actual requirements.

d) Code modifications are therefore made on the spot.

e) Under the intense pressure of commissioning a large system, the code begins to diverge from the documentation.

3. The Eureka Moment

This occurred during the third glass of wine after the second day of EuroForth 2004.

The documentation essentially contains the same information as the code itself. The only difference is that the documentation refers to names of sequences, steps and signals, as well as numbers. However, the documentation already contains a list of signals, therefore sequence and step names is the only additional data in the documentation.

Although each sequence is different, the same phrases e.g. "Hoist goes up" are frequently repeated from job to job. They already exist in French and German. Hence automatic translation is usually possible.

Therefore, the code can be used to generate, and even translate the code, provided only two new words are added to the wordset:

```
n SQ" Sequence name"     \ Document sequence name, then do SS CASE
n ST" Step name"         \ Document step name, then do OF
```

4. The specification

As we developed the specification, we realised that further advantages could be gained. Since the code had generated the documentation, the program "knew" about the documentation structure and thus could display the information dynamically, potentially making the debugging of sequences much easier. For example, the current step could be described, and each logical action could indicate its true or false state. To find a faulty signal, it was merely necessary to notice which one was coloured false. The visualisation therefore became an important feature of the new sequence system.

However, the overriding requirement was that, with the exception of the two words mentioned above, the code should be exactly the same as before - just as simple and compact.

5. The implementation

Data about a sequence is generated during compilation. The function assigned to the word differs for different stages of the compilation, so every word that needs to generate documentation has to be be declared as DEFER-ed.
Example:

```
DEFER X ( x---f )
DEFER ?NS ( f ---)

: ~?NS ( f--)           \ "~" sign added to the word's name
  IF 0T NS THEN ;

: ~X ( x---f )          \ True if input is ON
  INPUTIMAGE + C@  ;

: IMM-X
  GET-FROM-DP DROP      \ Gets compiled input number from dictionary
  COMPILE ~X ;

: IMM-?NS
  GET-FROM-DP DROP      \ Puts "next step" entry into data structure
  COMPILE ~?NS ;
```

Every new word which needs to generate a description is registered (added) in function GET-SQD-WORD, to permit automatic assignment of multiple words during invoking words: START-DESCRIPTION and END-DESCRIPTION. START-DESCRIPTION sets IMMEDIATE flag of deferred word, END-DESCRIPTION resets that flag and assigns controlling words instead of compiling words.

Word GET-FROM-DP takes the last entry from the dictionary and the function "name" that called it and passes it to the word SQD-VOCAB, which fills the step data structure to preserve parameters.

```
STRUCT STEP-ELEM            \ Data structure for one step instruction
  CELL FIELD .DP_NUMBER     \ Any number associated with data?
  CELL FIELD .PHRASE-FUN    \ Points a function generating description string
  CELL FIELD .TYPE          \ Extra parameter to distinguish ie input from output
  CELL FIELD .PHRASE-ADD    \ Anything to stick on the end, holds a sqphrase number?
  CELL FIELD .TICKNUM       \ Store the number to recall the name (and CFA)
  CELL FIELD .TICKTYPE      \ Store the type of the word – number of parameters.
END-STRUCT
```

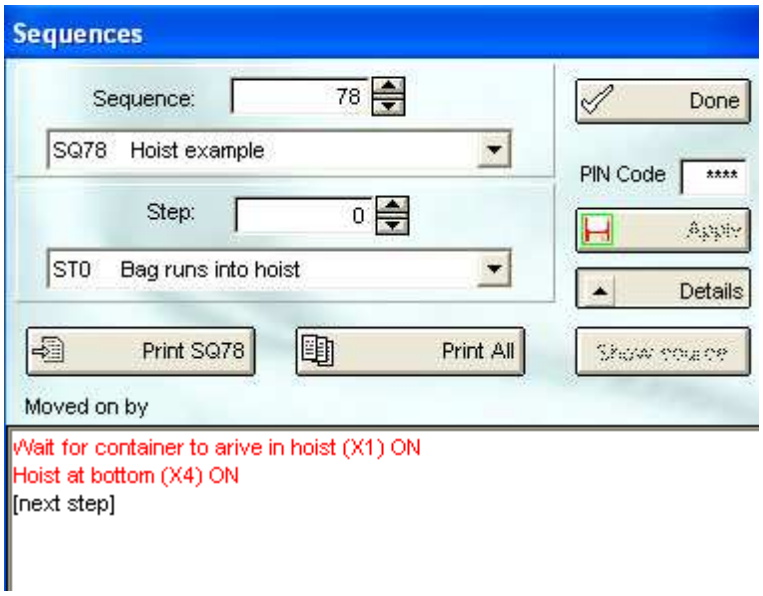
The list of instructions for each step of each sequence is generated, which index gives access to the instruction's parameters, description and FA.
A useful word SOURCE>INT returns an integer that comes after a word in the source stream. So having a piece of code like this:

```
3 PARAM@ 5 =
```

we can get both numbers 3 from the dictionary and 5 from the source. -1 is returned if a word is next instead of a number.

6. The result

```
START-DESCRIPTION
: SQ78
  78 SQ" Hoist example"
    0 ST" Bag runs into hoist"
          1 X 4 X AND ?NS          \ IO names taken from IO data base
    ENDOF
    1 ST" Bag settles in hoist"
      3 SECS ?NS
    ENDOF
    2 ST" Hoist goes up"
      2 X ?NS
    ENDOF
    3 ST" Bag runs out of hoist"
      3 X 20DNS
    ENDOF
    4 ST" Hoist goes down"
      4 X ?OS
    ENDOF
;
END-DESCRIPTION
```



7. Conclusion

The new system has already been used in three installations and has greatly decreased the time taken to write and debug sequences. It has also proved very popular with the plant maintenance technicians, who can identify faulty mechanical parts, sensors or actuators more quickly.

Simplicity in Forth

Federico de Ceballos
Universidad de Cantabria

federico.ceballos@unican.es

October, 2005

Abstract

In his book "Simplicity" [1], celebrated author Edward de Bono (famous for concepts such as "Lateral Thinking" or "Po") put forward ten rules that should be used in every system that tries to define itself as simple. This paper studies how the Forth language meets these rules.

1 Advantages of Simplicity

Simplicity is a nice word and, therefore, one we would like to have at our side. Simplicity can ease our lives and our actions. Learning a simple system saves us time, money and energy.

Simplicity is both elegant and powerful.

A complex system may have *the user's illusion*. This means that the user believes he or she commands the system and is in charge or everything. However, this can be very far away from the truth.

2 Disadvantages of Simplicity

By travelling the simplicity way, we may end being simplistic. By doing so, we may lose the usefulness of the original idea.

De Bono mentions the following advantages of a very complex book:

> *If you don't have anything to say, it is better if you say it in the most complicated way possible, otherwise the other people will notice the lack of content.*
>
> *Critics will love your book because they will feel as privileged, as they believe to be the only ones who can understand it.*
>
> *Critics will use rivers of ink describing your work, something that doesn't happen with a simple book.*
>
> *University professors will appreciate the book, because their science will be needed in order to explain it to the common people.*
>
> *Nobody will dare criticize it, as nobody will feel sure about having understood it.*
>
> *Philosophers may read in the book whatever they like, as its complexity justifies any interpretation.*
>
> *The public will buy the book to show off their own culture, even if they don't ever read it.*
>
> *The book will become a cult object.*

It will be natural to think that the author is a profound thinker studying very complex concepts.

A lot of fake intellectuals will have a good time, enjoying the complexity of the book.

These don't really apply to software, as the view from the end-user will not usually include the view from the inside. It may happen that a really complex system may be thought of as a marvel of simplicity.

3 How to Search for Simplicity

The following strategies can be used in order to advance in the quest for simplicity.

The historical analysis

Cut

Listen

Combine

Find out the concepts

The « mass » and the exceptions

Restructure

Begin again from square one

Make modules and smaller units

The provocative amputation

Wishful thinking

The energy transfer

The ladder approach

The perfume approach

Even if they are general concepts, a programmer will find that these phrases evoke methods that he or she has used in the past.

The idea of modular programming with short words should be close to any Forth programmer's heart. On the other hand, the idea of *combination* to generate the answer to several problems at the same time will probably bring forward the over-generalized solution criticized in Thinking Forth [2].

4 The Ten Rules of Simplicity

In this section, we shall take a detailed look to each of the rules prescribed by de Bono, not from his point of view, but rather from their usefulness to a Forth programmer.

You must attribute a high value to simplicity

Forth is a useful language. However, other computer languages can be regarded as equally useful. It may even be argued that the usefulness of mainstream languages is greater than that of "niche" ones.

A Forth programmer should therefore be conscious about the different among different paradigms and the reason behind the choice.

You must pursue simplicity with determination

It is not often the case that there is only one way of solving a particular task. According to these rules, the simplicity of the solution should be one of the factors taken into account when comparing different approaches.

You must thoroughly know the soil you are treading

One thing many Forth programmers have in common is the intimate knowledge (the phrase "carnal knowledge" comes to mind) they have about the programming environment they are using. They also extend this intimate knowledge to the hardware they are using and sometimes even to the external system controlled by the hardware.

All this should give this kind of programmer a head start when approaching a new problem.

You must project alternatives and possibilities

The Forth language encourages the programmer towards an incremental development. This way, the programmer improves his or her knowledge of the problem as he or she advances in the solution.

It should be noted that the best way is often unknown until the end of the project is getting closer.

A language that allows the programmer to easily compare different alternatives using a level of development that can be shown to work correctly is an interesting choice.

You must discuss and eliminate some of the existing elements

In a normal language, the resources available are by and large fixed. What you can do is extend the language with some given packages. Sometimes these packages are just black boxes ready to be used and other times the user can customize them.

In Forth the compiler can be enhanced, changed and sometimes even built again from scratch.

The additional packages are nearly always given as source code, divided into minute words that can be included only when needed and can also be changed easily.

Because of this, the author has chosen Forth when he tried to develop a simple environment that could be mastered by the end user in all its details [4].

This is a clear advantage, but it can transform itself in a problem and the programmer spends a lot of time "improving the tools" and no much is left to really solve the problem at hand.

Another problem is that we can be carried over by the sheer beauty of elimination. However, when we eliminate everything, nothing remains [6].

You must be ready for a fresh start

In his book "*The Mythical Man-Month: Essays on Software Engineering"* [3], Frederick P. Brooks argues that in many cases a full system should be developed with the only aim of using it to learn how that sort of system should be properly developed. Once this first version is ready, it should be thrown away and a second one should be started from the beginning.

However, this statement should be taken with a grain of salt. If the system is really complex, too much expense would already have gone into it so it would be difficult to justify that time and money just for experience. (As the dictum goes, experience is what you get when you don't get anything else.) If the system is simple enough, a good programmer should be able to write at least some useful code from the very beginning.

In Forth, the programmer should have a nice toolset, and this should improve with the time spent in the project. If some part of these tools couldn't be used, we would think that something has gone terrible wrong.

You must use concepts

Forth is made of words. It could even be argued that Forth is nothing but words. The name of the word is an important part of any definition.

Choosing good names is an art in itself. This is more important in Forth than in other languages.

As Dijkstra put it: *Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer*.

It may be necessary to divide things in smaller units

The human mind is really good at analysing problems, far above that any machine developed so far. On the other hand, it is widely acknowledge that this same mind has some basic limitations when trying combine several different ideas.

According to Miller [5], seven is the magic number that matches the different concepts what can be kept in our heads at the same time. Forth allows the programmer to code using tiny definitions. It has been said that the correct length of a definition should be one or two lines long. (Maybe using seven other words, once the stack noise has been removed.)

You must be prepared to sacrifice other values in favour of simplicity

It would be naïve to think that Forth is the only language that can be used to solve a given problem. Furthermore, it would be simplistic to assume that other languages are in a different league and only we are using Forth because of some sort of arcane knowledge.

We must assume that other languages have some important advantages over Forth:

> The compiler is readily available when buying the machine or installing the operating system, so that it can be used without any especial action.
>
> A great number of book, articles, examples, tutorial and web pages are available.
>
> It has a great user base and knowledge can be shared.

Maybe these reasons are not very important to us. However, they are there, and we should know what sacrifices we are doing if we decide to use other way.

You must know in whose name you are projecting simplicity

This is a really difficult subject. It can be easily forgotten what we are looking for and which advantages we shall gain with it. It is easy to let yourself go in the beauty of problem solving and forget that our code and our computer are only tools in the way and not the final objective.

We could be tempted to describe Forth metaphorically, as Mike Ham did: *Forth is like Tao: it is a way, and is realized when followed. Its fragility is its strength, its simplicity its direction*. This sounds nice, but a language is also a tool. At the end of the day, we would like to be able to enjoy the result of our work and not only how much we enjoyed working.

5 Conclusions

The author believes that the language Forth should be presented to other programmers as an example of a simple language capable of meeting most if not all of their need in quite an elegant manner. Even if they keep using their favourite language, the insight given by Forth will be priceless.

On the other hand, Forth programmers should also ponder what the rest of the programming community is doing and how it is solving day-to-day problems more or less successfully. A Forth programmer should not be blinded by the shine of his or her tools and should restrict him- or herself to carry out the current task in the most efficient manner.

References

[1] Edward de Bono. *Semplicità*. Sperling & Kupfer Editori, 1998.

[2] Leo Brodie. *Thinking Forth*. Fig Leaf Press, 1984.

[3] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, 1995.

[4] Federico de Ceballos. *A Minimal Development Environment for the AVR Processor*. 17th EuroForth Conference. Schloss Dagstuhl, 2001.

[5] George A. Miller. *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*. The Psychological Review, 1956.

[6] C.H. Ting. *Tao of Forth*. Twenty-First Forml Conference. Asilomar, 1999.

XML, SOAP and Web Services in Forth

Stephen Pelc
Microprocessor Engineering
133 Hill Lane
Southampton SO15 5AF
stephen@mpeforth.com

Abstract

Web services enable applications to exchange data using an extension to HTTP. Implementing web services requires extensions to an HTTP server, parsing and generating XML and then interfacing to other applications. This paper discusses what was needed to extend MPE's PowerNet to handle web services, and how we took advantage of Forth itself to simplify the solution.

Introduction

Web services are a a means of exchanging data between applications. The transfers are designed for machine use, not for human use; despite this the transfers are mostly printable. Data is exchanged using XML templates and data descriptions. You can treat XML as an extensible version of HTML with stricter rules.

Unlike many other application interchange protocols such as DCOM, web services are based on open standards and so are not restricted to use under specific operating systems. Because all data transfers are in text form, web services do not suffer from the data marshalling issues of other techniques.This is achieved at the expense of an increase in data size of about 10:1, which can make severe demands on the underlying networks used to link machines. It has long been MPE's opinion that heterogeneous interoperability is most easily achieved using sockets and text transfers. This was an ideal opportunity to test the assertion.

The work was supported by Construction Computer Software (Cape Town, South Africa), and so follows their requirements. The initial design work and test systems were provided by Graham Stevenson of Oxford Network Solutions.

After much reading of documentation and specifications, the implementation order was:

- XML input
- XML output
- Testing against an existing web service
- PowerNet changes
- Generate WSDL files
- Test with Excel

PowerNet v3

PowerNet v3 is a Forth TCP/IP stack with Telnet, web server, CGI and ASP facilities. It has been running for some years on embedded systems. The version for Windows replaces the embedd TCP/IP stack with calls to the Winsock API. Above that level, the multi-threaded

servers require very little change between the embedded and the Windows versions. Scripting facilities are provided by Forth itself.

At a very early stage in the design of PowerNet, we decided to implement each connection to a server as a task, and to treat the TCP/IP sockets as standard Forth I/O streams. Although this can increase the amount of RAM required by a busy server, it has the big advantage of simplicity. An additional advantage is that the usual Forth I/O handling, particularly **KEY EMIT** and friends, can be used with each connection. This design decision was to pay off handsomely when implementing web services.

An example transaction

The following is a SOAP request to a server:

```
POST /service1.asmx HTTP/1.1
Host: oxns.demon.co.uk
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://oxns.demon.co.uk:37851/HelloWorld"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
>
  <soap:Body>
    <HelloWorld xmlns="http://oxns.demon.co.uk:37851/">
      <s1>string</s1>
      <s2>string</s2>
      <i1>int</i1>
    </HelloWorld>
  </soap:Body>
</soap:Envelope>
```

The response is generated from the script file *HelloWorld.aspx*:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
>
  <soap:Body>
    <HelloWorldResponse
      xmlns=http://oxns.demon.co.uk:37851/
    >
      <HelloWorldResult>string</HelloWorldResult>
```

```
    </HelloWorldResponse>
  </soap:Body>
</soap:Envelope>
```

The request's *SOAPaction* header is parsed to yield *HelloWorld*, which is the required action and corresponds to a Forth wordlist. This name is extended to select the file *HelloWorld.aspx* which is output and processed by the ASP processor with the selected wordlist in the search order. The ASPX file that generated the response above could have been as follows.

```
<?xml version="1.0" encoding="utf-8"?>
<% language=forthscript %>
<soap:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
>

  <soap:Body>
    <HelloWorldResponse
    xmlns="http://oxns.demon.co.uk:37851/">
      <HelloWorldResult>
        <% /s1 .Param ." :" /s2 .Param ." :" /i1 .param %>
      </HelloWorldResult>
    </HelloWorldResponse>
  </soap:Body>
</soap:Envelope>
```

The key line is this one:

```
<% /s1 .Param ." :" /s2 .Param ." :" /i1 .param %>
```

This text between **<%** and **%>** is treated as Forth sourced and **EVALUATE**d by the Forth interpreter. Any output from the Forth is simply sent to the socket to which the response text goes.

XML input design

XML is an enhanced HTML with much stricter rules. In particular, every opening tag must have a closing tag. The major difference is that the choice of tag names is up to you, and each section of an XML document forms a tree.

As with HTL, you can choose to ignore tag pairs. We chose to process input the tags at the closing tags because the text was available as the content of the tag.

```
      <s1>string</s1>
      <s2>string</s2>
      <i1>int</i1>
```

Thus, for output we need to be able to access the tags by name, and to extract the data from them. The simplest way is to create Forth words for tags we wish to process, and to ignore all others. In the output phase, we can then use these Forth words.

All Forth words corresponding to tags share a common data structure which controls how data is set and displayed. To ease construction of services, a source notation was devised. An example is:

```
[services
  [service Service1
    Xcstring: /s1
    Xcstring: /s2
    Xint: /i1
    Xint: /i2
    Xfloat: /fl1
    Xoperation: HelloComplexWorld
    Xoperation: HelloLong
    Xoperation: HelloWorld
    Xoperation: HelloInts
    Xoperation: HelloFloat
    Xoperation: HelloDouble
  service]
services]
```

Several services can be available from one server. A service description consists of the data it uses and the operations it supports.

XML parser implementation

We started from Jenny Brien's code published in ForthWrite, the magazine of the UK Forth Interest group. It was later reimplemented by Leo Wong, with extensions for handling attributes, which are the name/value pairs (`name="value"`) found after the tag names inside a tag declaration.

The code has been extensively rewritten to add error checking and to deal with more cases which were discovered when we exported a large Excel spreadsheet to XML.

The intention of the original code was to be able to **`include`** an XML file as a Forth source code file. This makes testing easy, but has limitations when dealing with web services as the HTTP headers have to be bypassed. The solution was to provide a version of **`include`** that we call **`IncludeMem`** `( caddr u -- )` which performs the function of include from a block of memory. This word has other uses in embedded systems and is sufficiently useful that we incorporated it into the VFX Kernel, not least because the word has carnal knowledge of the kernel.

The final code code can be found in the file *Lib\XML.fth* in all VFX Forth for Windows distributions.

XML output design

Output of XML for web services is defined in ters of standard data types. Having defined a structure for each data item, we can insert the correct output routine when we define an instance of a data type. In the ForthScript below

```
<% /s1 .Param ." :" /s2 .Param ." :" /i1 .param %>
```

the words **/s1 /s2** and **/i1** correspond to the closing XML tags `</s1> </s2>` and `</i1>`. The word **.Param** displays the data in XML format.

The only issues here are in matching the specification, and in converting the XML special characters such as the '<' and '>' characters to their XML representations.

Testing against an existing web service

The parser was tested by constructing an example web service using the Microsoft C#.NET toolchain. This showed what we can expect from other systems. We could then test Excel against this web service. By logging the transactions, we could see what was expected.

The objective of our first server was thus to replicate the test server.

Required changes to PowerNet

The major changes to PowerNet were in the detection of a web service request. We handled this by using the ASMX extension for web service files. Another change was that most web service requests are made as POST requests, whereas most web pages are served as GET requests. GET requests are used when the state of the server will not change. Since web services exchange and modify data, the state of the server can change, and so POST requests are used.

We also had to modify the CGI handler to recognise the "?wsdl" string which is discussed below.

WSDL files

It is all very well to be able to exchange data, but you also have to be able to publish **how** you are going to exchange the data. This is handled by the Web Services Description Language (WSDL, or "wizdl"). Every web service includes two files which can be accessed by GET requests.

The first is a standard web page which tells humans how to use the service and often includes software documentation. The second is an XML description of the service as an XML schema.

In the real world

So does it really work? Yes, it does. There were the usual problems with Excel not being totally standards compliant, especially in terms of requiring "keep-alive" connections. However, since these restrictions are the result of recommendations in later versions of the HTPP RFCs, these problems may be forgiven even if they cause problems in low-resource environments such as are found in embedded systems.

Embedded systems

Much as we have complained about the behaviour of Excel, it is in reality the application that most people want to be able to exchange data with. PowerNet v3 can be coerced to work with 16k of RAM in total, is much more comfortable in 32k, and surprising in 64k bytes. The effect

of keep-alive connections with Excel is that you have to generate XML output data (or at least know its size) before generating the HTTP header.. This requires either more code or more RAM. We simply chose the more RAM route for the PC implementation.

Future developments

We intend to port PowerNet v4 to embedded systems, and to enhance the ease of use by automating the generation of the response scripts and WSDL files.