Inside the MPE VFX code generator

Stephen Pelc
MicroProcessor Engineering
133 Hill Lane
Southampton SO15 5AF
England

Tel: +44 (0)23 8063 1441
Fax: +44 (0)23 8033 9691
Net: sfp@mpeltd.demon.co.uk
Web: http://www.mpeltd.demon.co.uk

The MPE VFX code generator for Forth produces code similar in quality to that produced by many C compilers. Previous papers at EuroForth 1998 and Rochester 1999 discussed the genesis of the VFX system. This paper discusses the internal organisation of the VFX code generator, and the results produced.

Introduction

Over the years, I have increasingly been told that interpreted languages are slow, and Java has done nothing to reduce this impression. Forth is held to be an interpreted language, but it would be truer to say that it is an interactive language. Rather than have to justify an interpreted solution, MPE decided to produce a Forth system which generates good quality code, while retaining speed of compilation and full interactivity.

The VFX code generator is incorporated in ProForth VFX for Windows and the upcoming ProForth VFX for Linux, and achieves a performance that is two to three times that of other commercially available Forth optimising compilers. The VFX code generator is also incorporated in the MPE Forth 6 VFX cross compilers for ARM/StrongARM, Intel i32 architecture, Hitachi H8/300H, Motorola 68xxx, Coldfire and 68HC12.

Objectives

The objectives are:

1) good code quality
2) low increase in code size over threaded code
3) portability of the code generator
4) maintainability of the code generator

Good code quality is necessary not only as a sales feature, but also because it reduces the proportion of the Forth kernel that has to be written in assembler, so reducing porting costs. The MPE VFX kernel typically has fewer than ten code definitions. From the customer's point of view, the code quality reduces the amount of assembler code that is needed, and for the embedded system developer, the code quality is perfectly adequate for all but the heaviest interrupt loads.

The requirement for only a small increase in code size is created by customers with previous versions of the MPE Forth cross compilers. They will not be pleased by significant increases in code size. In addition, good code size is important with embedded RISC processors that tend to have small caches. MPE recently ported 92,000 lines of code from a previous DTC (direct threaded code) system using the MPE Forth 5.1 68xxx cross compiler to the MPE Forth v6.1 VFX 68xxx cross compiler. The same hardware was used in both cases. The resulting code ran several times faster and was slightly smaller than the original code.

Good portability of the code generator to new target processors reduces implementation cost, both of the code generator itself, and in the reduction of target code that must be rewritten. In addition it should be easy to update the code generator easily when new features are added.

Maintainability of the code generator affects code reliability, lifetime costs, and ongoing development costs.

Complexity

Compared with the classical threaded Forth compiler, a code generating compiler is completely different in scale. However, every time a new target is created, the conventional code primitives have to be written, tested, and debugged. The tradeoffs are simply whether the resulting additional performace will attract a good price, and will significant cost reductions be available when writing a new target for a cross compiler.

After writing a number of targets for both RISC and CISC architectures, as well as for register limited CPUs such as the 68HC12, we can answer yes to both questions above.

Portability

Compared to writing a naïve subroutine threaded cross compiler with simple inlining, the MPE VFX code generator takes considerably longer to write, but this is balanced by the portability of the target code.

The VFX code generator is largely CPU independent, but variations in CPU architecture do affect it. Overall, the portability of the code generator is heavily dependent on how aggressive the optimisations are. The VFX code generator is aggressive, and so is affected by variations in CPU architecture such as

- number of registers available
- presence/absence of autoincrement/autodecrement addressing
- load/store or register/memory architecture
- cache architecture

Internal architecture of VFX

This being a commercial product, the amount of hard information that will be released is necessarily limited.

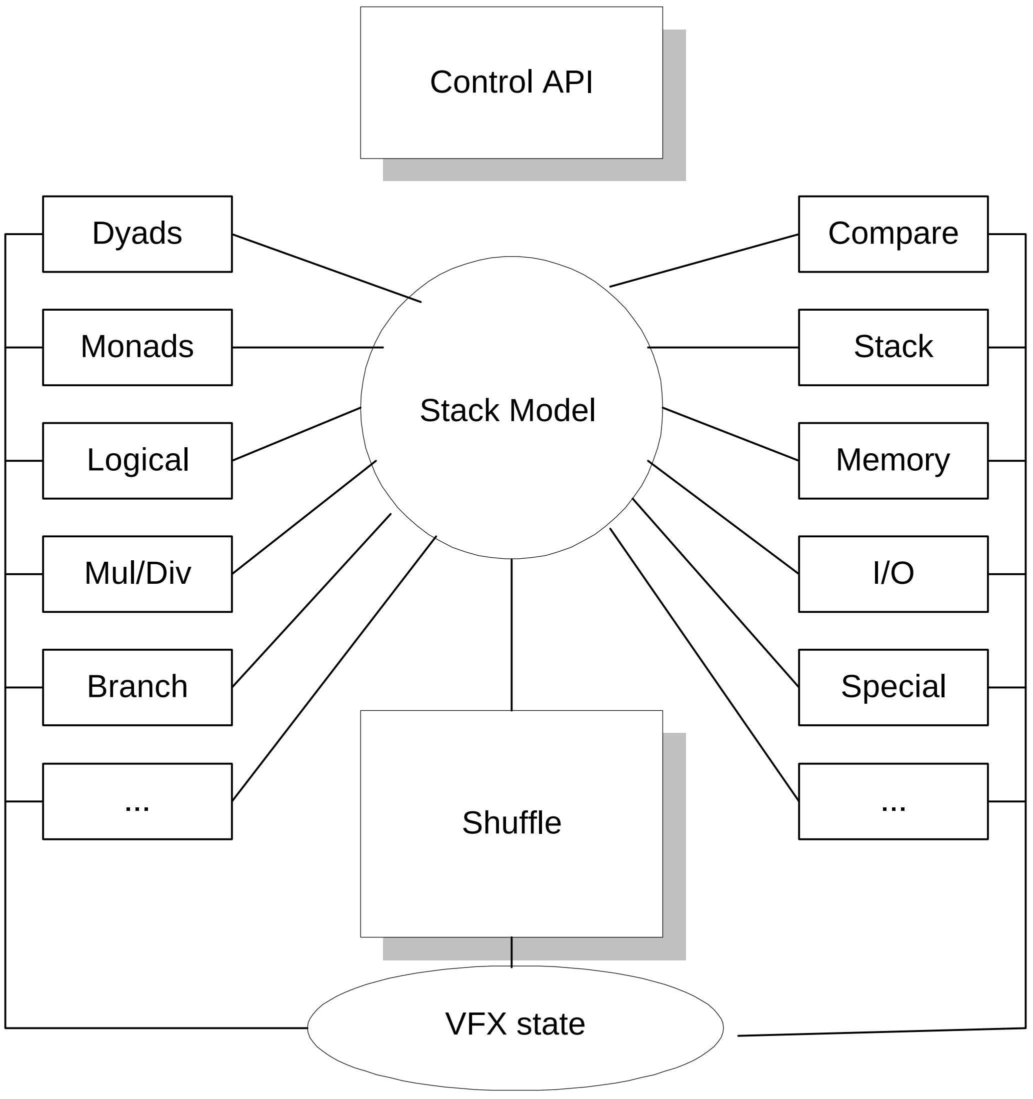
The optimiser defers code generation for as long as possible, and then processes the information, returning the stack at procedure exit to a canonical form. This compromise permits any optimised Forth word to be used in the same way as an unoptimised one.

The code generator is designed as one of number of phases from source to binary code. This topic is covered adequately in the compiler literature. At the outset, the code generator was envisaged as the middle of a three phase code generation sequence:

1. Token optimisation, e.g. source rewrite
2. Primary code generation
3. Reordering and peephole optimisations

At present, as will be explained later, the benefits offered by the first and last stages can be largely obtained by a few simple special cases in the code generator itself.

The VFX code generator can be broken into several major blocks:
Control API – how the system interfaces to the black box
Stack model – tracks the contents of the data stack
Stack shuffle – restoring the stack to a canonical state
Class generators – commutative dyads, monads, special cases
Special cases – literal folding, conditional branches …



Control API
The control API is the interface between the rest of the Forth system and the VFX code generator. Apart from the interface through the word's dictionary header, which includes a pointer to a code generator, and **`COMPILE,`** the other interfaces are concerned with controlling the level of optimisation and enabling/disabling various optimisations.

In cross compiler VFX code generators, the control API includes selection of the CPU version for those CPUs such the 68xxx family which have additional instructions in some versions.

Stack Model
The VFX code generator delays code generation until it has to lay code. This removes a vast amount of state tracking and code removal. Pure stack operations such as **`SWAP`** and **`DUP`** simply modify the stack model, keeping track of the resources used. This information is used by the class code generators to access the actual data. Rip up and retry code generation is treated as a special case, and for a typical CPU, only about ten such cases are needed.

Stack shuffle
In most processor architectures, it is beneficial to keep items in registers. In most VFX implementations, there is a canonical stack representation consisting of the top of the data stack in a register, the other items being indexed from a data stack pointer. This state is enforced on entry to and exit from a procedure and at other basic block boundaries. It is the job of the shuffle routine to restore this canonical state.

This routine is complex and the algorithm took several months development to be both portable and reliable. It affects nearly every other data structure in the VFX code generator.

Class generators
Many of the code generators can be grouped according to their function. For example, there is a group of operations which can be classed as "commutative dyads", which means that **a op** b is the same as **b op a**. Providing that the CPU instruction set permits it, all such code generators can use the same class routine with different parameters. Great use of defining words is made throughout the VFX code generator.

Several words, such as **`PICK`** and **`ROLL`** need special code generators. The number of such special cases is very dependent on the CPU instruction set.

VFX state
Apart from the stack model itself, VFX notes information such as the state of the return stack. On some CPUs such as the 68HC12 the last use of index registers is tracked. If a data item is removed from the return stack, the current definition is marked as not able to be inlined. This allows several run time actions such as for strings to be coded in high level Forth, and permits code as involved as Michael Gassanenko's BackForth extensions to compile without error under ProForth VFX for Windows.

This state information can be used internally within the code generator, and by external routines. Persistent information such as whether a definition can be inlined is stored in the dictionary header for the word.

Special cases
There are some special cases which are most easily handled by keeping state information and relaying new code under special circumstances. Examples of these are the code generation for the sequence "**`DOES> >R`**" and comparisons leading to a conditional branch such as "**`> IF`**". In the first case, the most efficient code sequence is different to that required if the **`CREATE`**

address is left on the data stack. In the second case, the code to generate an ANS "well formed flag" is redundant if a branch is to be used.

Heuristics

After the code generator has been built and tested, we then spend time looking at the code output under a range of conditions and coding styles. The disassembler is an integral part of the code generator! Inevitably we come across code sequences in client code that are less than optimal when compared with the same functionality written in MPE house style.

There are usually two reasons for these inefficiencies. One is because we have neglected a CPU peculiarity, the other is because a special case was neglected where further optimisation is available. Such conditions were particularly found with the code generator for the 68HC12, which has very few registers and limited 16 bit operations. After two or three passes over the code generator, we reached the state where our client stopped writing port access code in assembler, and reverted to writing high level code only.

Second level optimisations

Binary inlining
Short definitions can be copied inline under certain conditions. This a simple and widely used technique that avoids the call and return overhead. There are restrictions in its use, and in the recent VFX code generators for ProForth VFX for Windows, this technique has been largely superceded by source inlining.

Source inlining
Forth gains much of its power from very short definitions which are reused. Such definitions produce can produce dense code but do not avoid the cost of the canonical stack shuffle. Rergardless of whether or not structure definitions are available, we see a lot of code of the form:

```
: foo      \ addr n – n’
  2 cells + @ +
;


: boo      \ addr n – n’
  4 cells + @ +
;


: bar      \ addr – n
  0
  over foo
  swap boo
;
```

When compiled with only binary inlining under ProForth VFX for Windows v3.22 the results are as follows:

```
FOO
( 004922A8    8B5B08 )                MOV     EBX, [EBX+08]
( 004922AB    035D00 )                ADD     EBX, [EBP]
( 004922AE    8D6D04 )                LEA     EBP, [EBP+04]
( 004922B1    C3 )                    NEXT,
( 10 bytes )

BOO
( 004922D0    8B5B10 )                MOV     EBX, [EBX+10]
( 004922D3    035D00 )                ADD     EBX, [EBP]
( 004922D6    8D6D04 )                LEA     EBP, [EBP+04]
( 004922D9    C3 )                    NEXT,
( 10 bytes )

BAR
( 004922F8    8D6DF8 )                LEA     EBP, [EBP+-08]
( 004922FB    C7450000000000 )        MOV     DWord Ptr [EBP],
00000000
( 00492302    895D04 )                MOV     [EBP+04], EBX
( 00492305    8B5B08 )                MOV     EBX, [EBX+08]
( 00492308    035D00 )                ADD     EBX, [EBP]
( 0049230B    8D6D04 )                LEA     EBP, [EBP+04]
( 0049230E    8B4500 )                MOV     EAX, [EBP]
( 00492311    895D00 )                MOV     [EBP], EBX
( 00492314    8BD8 )                  MOV     EBX, EAX
( 00492316    8B5B10 )                MOV     EBX, [EBX+10]
( 00492319    035D00 )                ADD     EBX, [EBP]
( 0049231C    8D6D04 )                LEA     EBP, [EBP+04]
( 0049231F    C3 )                    NEXT,
( 40 bytes )
```

The binary inliner has expanded the code to 40 bytes. When the source inliner is enabled the results are very different. The definitions for FOO and BOO remain the same, but the code for BAR has changed completely:

```
BAR
( 004923C8    8BD3 )                  MOV     EDX, EBX
( 004923CA    8B5B08 )                MOV     EBX, [EBX+08]
( 004923CD    83C300 )                ADD     EBX, 00
( 004923D0    8B5210 )                MOV     EDX, [EDX+10]
( 004923D3    03DA )                  ADD     EBX, EDX
( 004923D5    C3 )                    NEXT,
( 14 bytes )
```

Even this can be improved by changing the register allocation strategy and special casing the code generator for “+”. This example illustrates the problems of special case or general register allocation on CPUs with a limited number of registers, usually considered to be eight or fewer. If registers are allocated too freely, the stack will have to be spilled more frequently, leading to larger and slower code. If they are not allocated too freely, cases such as the above will occur.

The **SourceInline** system allows factors to be processed as source code macros. Although it might be expected that these lead to an increase in code size, this has not been found to be true, especially when converting legacy code. Many of these small factors do simple things like add offsets. When compiled, either inline or as a call, the optimiser has to create the canonical stack form. After the call the optimiser then has to extract the items from the canonical stack. When processed as **SourceInline**, the full range of VFX optimisations is applied through the compilation of the factor.

Results

The code generation quality of ProForth VFX for Windows was compared with a number of optimising Forth compilers on the same machine.

```
Test time (ms) including overhead         VFX    iFth SF2.0  bigFor
DO LOOP                                    17      17    20    10
+                                          14      16    15    20
M+                                         25      46    45    55
*                                          19      24    36    25
/                                         108     151   104   130
M*                                         35      43    40    30
M/                                        114     110   105   165
/MOD                                      105     139   105   145
*/                                        150     130   145   165
ARRAY fill                                 35      77    95   110
=================================================================
Total:                                    665     791   720   855

Test time (ms) including overhead
Eratosthenes sieve 1899 Primes            906    1066  1070  1005
Fibonacci recursion ( 35 -> 9227465 )     827    1719   796  1455
Hoare's quick sort (reverse order)        621    2941  1825  3985
Generate random numbers (1024 kb array)   765    5163  4900  5065
LZ77 Comp. (400 kb Random Data Mem>Mem)  1045    5318  5375  5264
Dhrystone (integer)                       846    1231  3175  1160
=================================================================
Total:                                   5078   17490 17165 17956
```

The source code is available in the file BENCHMRK.FTH from the download area of our web site.

Conclusions

The MPE VFX code generator produces code that is between three and five times faster than the nearest equivalent commercial offering for integer benchmarks.

The code generator is portable and has been introduced in MPE’s ProForth VFX for Windows and in the MPE VFX Forth cross compilers.

There is an increase in development costs which is balanced by the performance results and the increased maintainability and portability of target code.

Acknowledgements

Some of this work has been done under ESPRIT project 25344 PRACTICAL

Papers by and conversations with the following people have been useful:
Gary Bergstrom, Paul Curtis, Anton Ertl, Peter Knaggs, Thomas Worthington.