Tradeoffs in the implementation of the Beetle virtual machine

Reuben Thomas

1st July 1996; revised 3rd June 2016

1 Introduction

Beetle is a virtual machine designed for running Forth compilers such as pForth [2]. It has been implemented in ANSI C [2] for portability, and, more recently, in ARM assembler. This paper describes the tradeoffs within and between the two implementations, and compares them to native Forth and C compilers.

2 The virtual machine

Beetle's computational model is a stack machine, and most of its instructions are zero-operand, taking their operands implicitly from the data stack. The only exceptions are branches, subroutine calls, and instructions to place numeric literals on the stack. There are about ninety instructions, most of which directly implement simple Forth words. A few others support specific control and data constructs.

The instruction set is encoded as a byte stream grouped into words of four bytes. Instructions are fetched a word at a time, and the bytes shifted out, decoded and executed. This makes instruction fetch more efficient than for a pure byte-stream encoding on many machines, though it introduces the overhead of testing to see when a new instruction word must be fetched. Since the instruction accumulator is arithmetically right-shifted after each instruction has been decoded, this is achieved simply by making opcodes 00h and FFh represent an instruction fetch.

Another consequence of grouping instructions into words is that branch destinations are word-aligned. Thus, gaps sometimes appear in the instruction stream where two control flows join; these are simply filled with 00h, so that when the gap is reached, execution proceeds immediately to the next instruction word.

The only major problem caused by using a word-stream design is that endism, or endianness, reared its ugly head. The solution adopted was to declare that byte addressing is little-endian, but that it does not matter how the bytes are actually stored. Thus, the only penalty is that byte addresses must have their two least significant bits inverted on big-endian machines; there is no penalty for word addressing, which is used for the majority of operations.

Address checking is optional in Beetle's design, because generally it is impossible to implement so that there is no time penalty when it is turned off, unless hardware checking is supported. Both implementations have optional address checking that can be configured on or off at compile time.

3 The implementations

A byte-stream-encoded stack machine is possibly the easiest sort of virtual machine to implement, and the change to a word stream only adds slight complication to the instruction fetch. The obvious implementation method, and that used in both implementations discussed here, is to switch on each instruction opcode to branch to the action routine for the instruction.

3.1 C Beetle

The main aim of the C implementation of Beetle was that it should be easily portable. To this end it was written in strict ANSI C, and implementation dependencies were carefully isolated. In this it was successful (to date it has been compiled on four different machine and operating system configurations), but it sacrifices speed. The only concession to efficiency was to have two versions of the interpretive loop, one for single stepping, and the other for continuous running. On the other hand, it was quick to write (the first version was completed in about three weeks, at a time when the specification was still changing), easy to debug, and served as a useful pattern for the other implementation.

The C implementation was provided with a simple command-line user interface, including such facilities as single-stepping, disassembly and simple profiling.

3.2 ARM Beetle

One of the design aims of Beetle was that it should be possible to produce fast hand-coded versions for particular machines. Such a version was written for the ARM processor (running under Acorn RISC OS). A branch table was used for instruction dispatch, and the routine for decoding the next instruction was expanded inline after each action routine. The registers of the virtual machine, such as the program counter and stack pointers, were mapped on to machine registers, and the action routines were carefully hand-coded. In particular, instruction decode was coded in three instructions and instruction fetch in one. Some optimisations, such as caching stack elements, were expressly forbidden by Beetle's specification, so that different implementations would be indistinguishable to programs running on them.

Address checking and single stepping were provided as independent optional extras. The single stepping mode also counts how many times each instruction is executed, to allow simple profiling.

The hand-coded Beetle is a direct replacement for the equivalent functions in the C version.

4 Measurements

Benchmarks were run on both versions Beetle running in all operating modes (that is, with and without address checking, and, in the case of the ARM implementation, with without single-stepping and profiling). These consisted of Forth programs compiled by the pForth compiler. For comparison, the benchmarks were also run on a native-code version of pForth, and some of the benchmarks were translated into C.

The main purpose of the benchmarks was to determine how closely interpretive performance approached compiled performance; other issues considered were the relative speeds of the virtual and real machines and the relative speed of naïvely-compiled Forth and optimised C (no optimising Forth compiler was available; however it is safe to say that it is unlikely to have been faster than the C compiler used, in any case).

All the benchmarks were run on an Acorn Risc PC 600, with a 30Mhz ARM610, in single-tasking mode.

System	Time/s	mips
Native ARM	4.3	-
Native Beetle (no checks)	5.6	1.1
Native Beetle (address checks)	8.0	0.76
Native Beetle (single stepping)	10	0.61
Native Beetle (checking & stepping)	14	0.44
C Beetle (checking off)	16	0.38
C Beetle (checking on)	26	0.23

Table 1: Timings for the ANSI test suite benchmark

System	Time/s	mips
Native ARM	20	-
Native Beetle (no checks)	35	1.9
Native Beetle (address checks)	54	1.3
Native Beetle (single stepping)	70	0.97
Native Beetle (checking & stepping)	88	0.77
C Beetle (checking off)	130	0.52
C Beetle (checking on)	200	0.34
GNU C (optimised)	2.0	-
GNU C (unoptimised)	6.5	-

Table 2: Timings for the primes benchmark (method 1)

The benchmarks were timed, and the number of interpretive instructions executed was counted. The static size of the benchmark and interpreter code was also measured.

4.1 The benchmarks

Two main benchmarks were run, one I/O-bound, and one computation-intensive. The first was a suite of tests designed to show that a Forth compiler complies with the ANSI standard. It is essentially a long script which is read from storage as the tests progress. The second benchmark was a pair of implementations of Eratosthenes's sieve, counting, in this case, all the primes up to 800,000, three times.

The code for the sieve benchmarks is given in appendix A; that for the ANSI test suite is long and unenlightening.

The results of running the benchmarks are given in tables 1– 4. All measurements were taken by hand with a stopwatch, and averaged over several readings, which rarely differed by more than 0.1s. All results are given to two significant figures.

When the number of instructions executed by Beetle were counted, only useful instructions were included; the instruction fetch instruction was omitted from the counts, and hence from the performance figures.

4.2 Code size

It is also interesting to compare static measurements of the different implementations of Beetle. Table 5 shows their sizes. The sizes of the C versions are those of

System	**Time/s**	**mips**
Native ARM	14	-
Native Beetle (no checks)	20	1.8
Native Beetle (address checks)	31	1.2
Native Beetle (single stepping)	39	0.93
Native Beetle (checking & stepping)	50	0.73
C Beetle (checking off)	75	0.49
C Beetle (checking on)	120	0.30
GNU C (optimised)	1.9	-
GNU C (unoptimised)	6.4	-

Table 3: Timings for the primes benchmark (method 2)

Benchmark	**Instructions / 10^6**
ANSI test suite	6.1
Primes method 1	67.8
Primes method 2	36.4

Table 4: Numbers of Beetle instructions executed in each benchmark

the object modules, and exclude the C runtime system and other support code; the sizes of the native versions includes all support code.

The benchmarks themselves were also measured. The ANSI test suite compiled incrementally as it ran into 5,372 bytes, and the two primes tests compiled into 356 bytes in total, with another 400,000 bytes used for the main array.

5 Analysis

The analysis is split into three parts; the first compares the two implementations of the interpreter, the second compares the hand-coded interpreter with the native pForth compiler, and the third makes comparisons between all three, and with natively compiled and optimised C.

Many of the comparisons refer to table 6, which gives the mean speed of each implementation relative to the fastest hand-coded version of Beetle; the tests run in C are also included.

Implementation	**Size/bytes**
Native Beetle (no checks)	3,920
Native Beetle (address checks)	6,692
Native Beetle (single stepping)	8,328
Native Beetle (checking & stepping)	11,100
C Beetle (checking off)	7,144
C Beetle (checking on)	13,864

Table 5: Sizes of different implementations of Beetle

Implementation	**Relative speed**
Native ARM	1.5
Native Beetle (no checks)	1.0
Native Beetle (address checks)	0.66
Native Beetle (single stepping)	0.52
Native Beetle (checking & stepping)	0.40
C Beetle (checking off)	0.30
C Beetle (checking on)	0.19
GNU C (optimised)	14
GNU C (unoptimised)	4.3

Table 6: Relative speeds

5.1 C Beetle *versus* ARM Beetle

There is no doubt that the hand-coded version of Beetle for the ARM processor gave a marked speed improvement: the slowest native version is 1.3 times quicker than the fastest C version; the fastest native Beetle is over five times faster than C Beetle with address checks. The difference between corresponding versions is a factor of about 3.4.

The only advantage that the native version of Beetle has over the ARM version is that it is easily portable; it required between thirty minutes and a day's work to port it to each of three different systems, while it took a week to write and debug the ARM version. Though a week is far less time than it took to write the C version originally, and not long for such performance gains, the present author did have the benefit of a thorough knowledge of Beetle, the ARM processor and the RISC OS environment; further hand-coded versions would certainly take longer to produce.

The code size of an interpreter is important insofar as the interpreter will run considerably faster if it can reside mainly in the processor's cache. The ARM610 has a 4kb cache, so this goal is only achieved by the smallest version of the native implementation; however, even within the interpreter there is locality of code, so that even the larger C version will have a reasonable cache hit rate. Typical high-performance microprocessors have much larger caches, in any case.

According to these considerations, it seems that hand-coding Beetle was a worthwhile effort on the machine in question; on higher performance machines the extra effort is only worthwhile if the extra performance is required; it is also not clear how much the performance would be improved on architectures which lend themselves less readily to hand-coding than the delightful ARM.

5.2 ARM Beetle *versus* native pForth

One of the most interesting results of the tests is that pForth runs only 1.5 times slower on the ARM version of Beetle than natively. This suggests that there is little point producing native versions of pForth at all, and that a good implementation of Beetle produces near-optimal execution of pForth.

An important point is that pForth is a naïve compiler, which performs no optimisation. Since Beetle is designed specifically to execute compiled Forth, a naïve compiler will produce good code for it; indeed, it is not possible to perform much optimisation, as Forth does not have conventional variables or scoping, and Beetle has no general purpose registers.

5.3 General comparisons

While the ARM implementation of Beetle compares well with a native Forth compiler, the comparison with optimised C shows that there is still a long way to go to obtain high performance. A factor of 14 is rather disheartening in one sense, but in another it is irrelevant, as pForth is not designed to produce fast code, but to provide a simple, comprehensible compiler, suitable for teaching the principles of Forth compilation. Forth, too, does not aim to give maximum speed, although Forth compilers usually provide assemblers so that time-critical code can be hand coded; often, only a small portion of a program needs to be coded to obtain C-like performance. Optimising Forth compilers also exist.

It is also worth noting that the test system is a slow computer. Even the C version of Beetle with address checking on runs acceptably fast on typical large multi-user systems; indeed it runs at much the same sort of speed as the hand-coded ARM Beetle. Native versions for such powerful systems would run even faster, and provide more than adequate power for typical program development.

6 Conclusions

The implementation of Beetle in ARM assembler demonstrates that resonable performance can be obtained from interpreters. Interpreters are also quick to implement, and easily portable, not much less so when hand-coded than when implemented in a portable high-level language.

Code compiled by a naïve compiler can execute at near-optimal speed when interpreted, provided that the interpreter is designed to support the language being compiled. There seems to be little point implementing simple-minded compilers in native code. It may even be a disadvantage for an interpreted compiler to perform optimisation, as this will increase compilation time, and in an interpretive environment, particularly one such as Forth, which makes use of incremental compilation, this will interfere with the rapid development cycle.

Even when implemented portably in a high-level language, interpreters can run at a reasonable speed; although they cannot use machine-dependent tricks, they can take advantage of optimising compilers to boost their speed.

References

[1] Jim Galbreath. A high-level language benchmark. *BYTE*, 6(9):180–198, 1989.

[2] Reuben Thomas. Beetle and pForth: a Forth virtual machine and compiler. BA dissertation, University of Cambridge, 1995. `https://rrt.sc3d.org/`.

A Code for the sieve benchmarks

These benchmarks are adapted from [1].

```
400000 CONSTANT SIZE
VARIABLE FLAGS SIZE ALLOT

: DO-PRIME   ( method 1 )
   FLAGS SIZE 1 FILL
   0
   SIZE 0 DO
      FLAGS I + C@ IF
         I 2* 3 +  DUP I
         BEGIN  DUP SIZE < WHILE
            0  OVER FLAGS +  C!  OVER +
         REPEAT
         DROP DROP 1+
      THEN
   LOOP
   .   ." primes " ;

: DO-PRIME-HI   ( method 2 )
   FLAGS SIZE 1 FILL
   0
   SIZE 0 DO
      I FLAGS + C@ IF
         I 2* 3 +  DUP I + SIZE < IF
            SIZE FLAGS +  OVER I +  FLAGS + DO
            0 I C!
            DUP +LOOP
         THEN
         DROP 1+
      THEN
   LOOP
   . ." primes " ;
```