An Introduction to the Beetle Forth Virtual Processor

Reuben Thomas

24th November 1996

Abstract

Beetle is a virtual processor designed for the Forth language. It uses a modified byte-stream code designed for efficient execution which is binary portable between implementations. It has been implemented in C and assembler. The C implementation is completely machine-independent with the exception of interactive input and output; the assembler version runs the supplied Forth compiler at up to half the speed of the corresponding native code compiler and generates more compact code. Beetle is designed to be embedded in other programs; a simple debugger has been written to demonstrate this ability. Beetle can be configured to perform bounds checking on all memory references. A standard I/O library is implemented; access to native code routines is also possible, allowing Forth and C programs to call each other.

1 Introduction

Ever since the invention of high-level languages, one of the most popular implementation methods has been to use a virtual processor, often called an interpreter [4]. A virtual processor, or "VP", usually resembles a real processor, reading a virtual instruction stream and calling appropriate routines to carry out the actions of the virtual instructions. VPs are often designed to support a particular language, so that once a VP has been produced writing a compiler is straightforward, and porting the compiler requires little more effort than porting the VP. This method is commonly used for functional languages such as LISP and Miranda [3, 8], and especially for prototyping new compilers [5, 1].

Beetle is just such a VP, and the language it supports is Forth [2]. Forth is a simple stack-based language which is most commonly used in embedded systems; however, Beetle is more suited to general applications, Forth compiler development and teaching, as it lacks the speed and low-level hardware access required for embedded systems.

Add to the discussion above the fact that Forth is often implemented on top of a virtual machine [7], and it is hard to see what could be interesting or novel about Beetle. The rest of this paper is an attempt to show that Beetle is worthy of interest, but before proceeding to describe it, three points should be made. First, while none of Beetle's features is revolutionary, they have not to the author's knowledge been combined before.[1] Secondly, virtual processors are often forgotten as they lie buried under the compilers they were designed to support, and there are few detailed reports available on the design and performance of specific VPs ([4] contains a good

[1]The same is true of the much more illustrious Java virtual processor; indeed, the fact that the techniques involved in its construction are all tried and tested is its greatest strength.

bibliography of those which do exist).[2] Thirdly, after a period of quiescence, VPs are undergoing something of a renaissance, and must be reassessed in the context of the new uses to which they are put and machines on which they are run.

2 The virtual processor design

Beetle's architecture is similar to that of a real processor. It is stack-based, and all computation takes place on the data stack, so it has no general-purpose registers. There is also a return stack. All instructions are represented by one-byte opcodes. Instructions take either zero or one operand.

2.1 Memory

The memory is an array of four-byte words. The bytes in a word may be stored in either little-endian or big-endian order, so word addressing is always efficient, but byte addressing is little-endian, so that it is identical in all implementations. The penalty is a single machine instruction on big-endian machines to invert the bottom two bits of the address of a byte reference.

2.2 Registers

Beetle has a program counter, `EP` ("`Execution Pointer`") and two stack pointers: `SP`, the data `Stack Pointer`, and `RP`, the `Return` stack `Pointer`. `I` holds the current `Instruction`, and `A`, the instruction `Accumulator`, the next few instructions to execute. There are several other more specialised registers.

2.3 Execution

When Beetle is started, it performs the following execution cycle:

```
begin
    copy the least-significant byte of A to I
    shift A arithmetically 8 bits to the right
    execute the instruction in I
repeat
```

This demonstrates Beetle's main adaptation for efficient execution on modern processors, which is to load instructions severally rather than singly. `A` is four bytes wide; on most modern processors it is at least as fast to load a four-byte word from memory as to load four single bytes. When the accumulator becomes empty the value zero or 255 is copied to `I`; these are the opcodes of the instruction `NEXT`, which causes the word pointed to by `EP` to be loaded into `A`.

It might seem that `NEXT` would therefore be executed at least every fifth cycle, but since instructions such as branches perform an implicit instruction fetch it is in fact only executed as about 10% of instructions obeyed.

2.4 Operands

The only operands are numeric literals and branch addresses. Where possible these are packed into the instruction word directly after the instruction opcode; addresses are turned into offsets for compactness. Such immediate operands always occupy

[2] Recent literature includes interesting reports on new types of virtual processor [6], but while these are innovative and worthy of note, they are mainly relevant to researchers in their application areas, and not to VP designers in general.

the rest of the instruction word. If the operand is too big, then it is placed in the next available word; further instruction opcodes may still be placed in the current word.

In the execution cycle `A` is shifted arithmetically rather than logically. This allows negative literals and branch offsets to be used without needing extra opcodes. By shifting `A` before the instruction is executed immediate operands are accessible to their instructions with no further decoding required.

2.5 Implementability

Because of its simple design which uses quantities no smaller than a byte and no bigger than a four-byte word, Beetle is easy to implement, whether in a high-level language or assembler. In a high-level implementation, Beetle's registers map obviously on to variables and the memory can be represented as a byte array, manipulated with array operations. All these operations are optimised well by optimising compilers. In assembler Beetle's registers map naturally on to machine registers, and ordinary memory addressing instructions can be used to manipulate its memory.

The non-recursive design means that static allocation techniques can be used, which keeps assembler implementations simple and makes them more likely to be correct. The use of twos-complement arithmetic again leads to a natural and efficient implementation; few computers still use other forms of arithmetic.

3 Portability

The greatest benefit of Beetle's implementability in high-level languages is that it can easily be made portable. This was the main goal of the C implementation, which is written entirely in ANSI C, and uses the standard libraries almost exclusively. The only exception was forced by the nature of Forth: since it is interactive, it requires unbuffered character input and output. These can sometimes be achieved using the ANSI libraries, but it is not guaranteed; on most operating systems (apart from UNIX) simple routines are available to input and output single characters.

Thus character input and output macros must be supplied along with the endianness and a few other machine characteristics for each machine on which Beetle is to be compiled. The configuration is isolated in a special header file, several of which have been prepared for various operating systems. Special arrangements are made for UNIX, and extra code is supplied to implement unbuffered input and output.

C Beetle has to date been compiled and tested successfully on five different operating systems, including three versions of UNIX.

Beetle's simple design also makes it easy to write hand-coded implementations quickly: the ARM version was completed in a week. This is still a short time to transport an entire Forth system to a new environment, and as will be seen in section 6, it gives markedly better performance.

4 Compiler support

Conventional Forth compilers use indirect threaded code, in which Forth words (the equivalent of functions or procedures in other languages) are compiled as lists of addresses pointing to other words. Each word has a code field, which contains a pointer to code to execute the word; in most words, this is the address interpreter, a minimal VP which interprets the list of addresses, finding their code fields in turn and branching to them. In primitive words written in assembler the code field will point to the corresponding machine code.

This VP is so simple that it consists of just a few instructions. It is flexible in that it can be adapted to many designs of compiler (and indeed, many languages). On the other hand its code is not portable, nor particularly dense, as even primitive instructions such as those provided in Beetle's instruction set occupy a machine word, generally four bytes on modern processors.[3]

By providing a range of specialised instructions (most of them corresponding directly to ANSI Standard words), Beetle allows compact and portable code to be generated. It also simplifies the implementation of compilers, at the expense of fixing some design choices such as the number of stacks, and forcing the use of the return stack for loop indices and counts.

Beetle provides most of the ANSI Core Word Set arithmetic, logical and memory access words as native instructions, and directly supports all the usual Forth looping constructs, as well as the `CREATE...DOES>` datatype declaration mechanism. Support is also provided for exceptions, though not for local variables. This practice of providing instructions with relatively high semantic content lessens the overheads of interpretation, as a lower proportion of time is spent fetching and decoding instructions.

5 Embedding and safety

Beetle is designed to be used as an interpreter embedded in other programs. The C implementation provides a header that programs may include to use Beetle; at the moment only one instantiation of the interpreter is allowed, but this restriction could easily be lifted. The memory and all the registers are available to the program to be inspected and manipulated, so the C program can both control Beetle, and, by means of Beetle's `LINK` instruction, be called by it, arguments and return values being passed on Beetle's data stack. The facilities provided are enough to write a debugger, and a simple debugger was written to aid the development of the C implementation of Beetle and the porting of the Forth compiler to Beetle.

One of Beetle's registers, `CHECKED`, controls whether address checking is performed; in the C and assembler implementations its value is fixed at compile time. When enabled, address checking is performed on all memory references, and out-of-bounds and unaligned memory references are trapped and reported to the calling program. In this situation, Beetle cannot corrupt the program that controls it directly, though it can cause corruption or a crash by indiscriminate use of `LINK`.

These features make Beetle a good candidate for an embedded interpreter, whether to implement an application-specific scripting language, or to provide a safe way for an application to generate and run code on the fly.

6 Performance

Beetle cannot hope to outperform native machine code because of the interpretive overhead. The most important question to address is whether it runs fast enough; it is also interesting to see what the interpretive slow-down is.

Three main benchmarks were run: two computation-intensive prime-finding programs, and a compiler testing program, which was read from disk as it progressed,

[3]It is often claimed that Forth translates into particularly dense object code, but this is doubtful on modern machines. One explanation given is that Forth programmers simply write smaller programs than programmers using other languages; it should also be borne in mind that Forth has a much bigger advantage over other languages on 8-bit processors, on which addresses are only sixteen bits long, and relatively much denser compared with machine code than on 32-bit or 64-bit processors.

and also exercised the input-output functions quite heavily. The timings discussed below are those taken when Beetle was peforming address checks.

The timings were taken on a machine with an ARM610 processor rated at about 20mips. The C implementation of Beetle ranged between about 0.4mips and 0.5mips, and the native implementation averaged around 1mips; the raw interpretive slow-down is between 20 and 40 times. The actual slow-down was much less: when the benchmarks were run on the native ARM version of the Forth compiler, they ran only 2.3 times faster than native Beetle, and 7.9 times faster than C Beetle. It should also be remembered that Beetle input-output operations, such as reading bytes from a file, count as a single interpretive instruction.

The tests also completed in a reasonable time on the native Beetle, though on C Beetle they were rather slow, taking 200s to find all the primes up to 800,000, compared with 54s for native Beetle, 20s for a native Forth compiler and 2.0s for a C translation, compiled with optimisations. Only Beetle performed address checks, and the Forth compilers did not optimise; indeed, Beetle's virtual code cannot be optimised much. This is an advantage inasmuch as a naïve compiler will produce near-optimal code; however, since the instructions are low-level, many are executed (67.8 million in the benchmark under discussion) and the interpretive overhead remains high. This contrasts with languages such as APL, in which VPs compete with native compilers for speed since they spend far more time performing instructions than decoding them [4].

Nevertheless, Beetle is more than adequately fast for interactive program development in the usual Forth style; since its compiler is ANSI-compliant, demanding programs developed on it could be recompiled with an optimising compiler to obtain better performance.

7 Conclusion

The Beetle system allows Forth programs to be developed in a safe environment. With the addition of some more input-output primitives, and extra instructions to support floating point arithmetic and the few other unsupported parts of the ANSI Forth Standard, the compiler could easily be extended to provide a full Forth development environment. Programs are instantly portable to a wide range of machines, and ports to new machines are simple. With a little more work, execution can be dramatically improved by hand-coding the VP. Interworking with C and other high-level languages is also straightforward, and Beetle can be embedded in application programs to drive a command language or run code generated on the fly.

Beetle's design is simple, but closely adapted to modern processors. It is easy to implement in high-level languages or assembler, and performs well in both cases. At the beginning of a new era of VPs, Beetle is a useful starting point.

References

[1] M. Alfonseca, D. Selby, and R. Wilks. The APL IL interpreter generator. *IBM Systems Journal*, 30(4):490–497, 1991.

[2] Leo Brodie. *Starting FORTH.* Prentice-Hall, second edition, 1987.

[3] Harold Carr and Robert R. Kessler. An emulator for Utah Common Lisp's abstract virtual register machine. In *Proceedings of the 1987 Rochester Forth Conference*, pages 113–116, Rochester, NY, 1987.

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

[5] Marc Feeley and James S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Proceedings of the 1990 ACM Conference*, pages 119–130, Nice, France, 1990.

[6] Richard M. Fujimoto. The virtual time machine. In F. Leighton, editor, *Proceedings of the 1989 ACM Symposium*, pages 35–44, Santa Fe, Mexico, 1989.

[7] R. G. Loeliger. *Threaded Interpretive Languages: Their Design and Implementation.* BYTE Books, Peterborough, NH, 1981.

[8] David A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.