Mite: a fast and flexible virtual machine

Reuben Thomas*
Computer Laboratory, University of Cambridge

15th August 1998

Abstract

Recent interest in virtual machines has been due mainly to the need to distribute executable content over the World Wide Web, where security and standardization are the key concerns. To fulfil the potential of distributed network computing virtual machines must be built into the OS kernel where speed and flexibility are of the essence. Mite is a virtual machine designed to allow optimising compilers to produce compact portable binaries which can be quickly translated at load-time into fast code. Its minimal core can be extended with support for language, machine or OS-specific features. Mite's design and current and planned implementation work are described, along with reasons why it should be of particular interest to Forth programmers.

1 Introduction

Virtual machines have existed for almost as long as computers. A microprogrammed processor is a hardware implementation of a virtual machine, and VMs have been widely used as compiler targets, whether to make the compiler easier to write or easier to port, or to increase code density, as with Forth. In recent years the explosive growth of the Internet and in particular the desire to add executable content to the World Wide Web has led to renewed interest in VMs. Since the prime target has been the consumer market, standardization and security have been the key concerns. This climate has given rise to several VMs with associated portable binary formats, such as the Java Virtual Machine [1] and Dis [2].[1]

These VMs are more or less tied to the operating environments with which they are bundled. Java's instruction set is heavily tailored to the Java language, though ingenious programmers have managed to compile languages as diverse as Forth[2] and ML for it, and the Java security and module system is built into the VM. Dis too is tailored to its operating system, Inferno, though it is more language-independent.

Existing VMs also tend to produce inefficient native code. Java's simple stack-based architecture requires much effort from a just-in-time compiler to produce even reasonable code, although Dis's three-operand memory-to-memory architecture is more CPU-like. Both lack information about the liveness of quantities and the location of loops which optimising native code compilers use to produce good code. Such information cannot be used by a compiler producing virtual code, which does not know the characteristics of the target machine, so it must be placed in the virtual code if efficient native code is to be produced.

*Reuben.Thomas@cl.cam.ac.uk

[1] Other existing VMs such as Taos and Omniware seem to give better performance and be language-independent, but information about them is hard to come by, seemingly for commercial reasons.

[2] See `comp.lang.forth` for information about various efforts in this direction.

Why are speed and flexibility important? To allow distributed network computing to fulfil its promise the VM must be built into the OS kernel, so that all services and applications can be distributed. But if all the code in a system is virtual, the VM must be extremely efficient. It must also be a good target for any compiled language, and be capable of interfacing with the parts of the system that still use compiled or hand-written native code, perhaps device drivers or the virtual memory system. This is the niche that Mite attempts to fill.

2 Aims, benefits and disadvantages

Mite is a standalone virtual machine, not tied to a particular operating system. It attempts to address the deficiencies in existing VMs outlined above. Inevitably, it substitutes different deficiencies, but the hope is that the benefits outweigh the costs. Mite aims to be:

Minimal Nothing more than a hardware abstraction layer for processors.

Efficient It should be possible to generate native code from virtual code whose performance is close to that of good compiled code with little effort in the JIT compiler, provided that the virtual code is of good quality.

Extensible It should be possible to extend the VM without affecting the core semantics.

Universal The VM should be a good target for any language currently compiled into native code. It should be possible to interpret or compile the virtual code.

The fulfilment of these aims brings the following benefits:

Familiarity As the VM should look similar to CPUs and hence familiar to programmers, the implementation of both high-level language compilers and native code generators from the VM should be simpler than for a less familiar VM model.

Adaptability It should be straightforward to add features required for particular systems, for example security features or concurrency primitives.

Some disadvantages are:

Poverty In its bare form, Mite is nothing more than a portable assembly language. For most practical applications it needs extending.

Reliance on good compilers Though it will be capable of producing good code, there is nothing to prevent naïve compilers producing poor virtual code, at the expense of poor native code.

3 The virtual machine model

Mite has a load-store RISC-like architecture. This maps well on to modern RISC processors, because it is easier to map registers to memory than vice versa: information about liveness and priority of quantities which is unnecessary when they are kept in memory is vital for good register allocation.

3.1 Registers

The crucial difference between Mite and other VMs is the way it handles registers. General-purpose registers are created as necessary and destroyed when they are no longer needed. The directive `NEW` n_t, p creates a new register which is referred to by its number, n. It is inserted after position p in the register list. `t` is the register's type, and denotes a temporary quantity. After creation a register can be moved within the list, and, at the end of its life, destroyed.

The register list is the lynchpin of Mite's register allocation scheme. An unlimited number of registers is not hard to cope with: excess registers are easily mapped on to memory. By keeping the registers in a list it is easy for a JIT compiler to perform register allocation: if k physical registers are available, then list positions 1 to k are mapped to physical registers, and the rest to memory. When the list is permuted so that registers move across the k boundary, spill and restore code is generated. Provided the register list is kept up-to-date to reflect the priority of the quantities in it, a good register allocation will be achieved regardless of the value of k.

3.2 Immediate quantities

Immediate quantities are even more restricted in Mite than in most RISC processors: the only place that they can occur is in `NEW` directives. A register may also be declared constant by replacing `t` by `tc`; it can then be treated, where possible, as an immediate operand, and need never be assigned to a physical register. On the other hand, by forcing all immediate quantities to be assigned to virtual registers, all the information needed for a good allocation to physical registers is available if it is required.

3.3 Register size and addresses

To be usable on both 32-bit and 64-bit architectures, Mite does not have a fixed register size, but requires registers to be the same size as addresses, as on most real processors. Addresses must be either four or eight bytes wide. The width of addresses is denoted W_A. In the assembly language, constants are two dimensional: the value of x+y is $x + W_A y$. Thus, on a machine with 32-bit addresses, the number 4+2 would evaluate to 12, whereas on a machine with 64-bit addresses it would evaluate to 20.

3.4 Memory model

Mite has a "pseudo-Harvard" architecture: code and data addresses may be in separate address spaces or in the same space. This not only accommodates Harvard architecture machines, but also allows more secure systems in which code is not addressable by data instructions, such as Inferno, to be built on Mite.

Memory may be addressed in groups of one, two or four bytes, or in address-sized groups. Memory accesses must be aligned to the size of the quantity being addressed. This is a good example of Mite's highest common factor approach: some processors can access unaligned quantities, but to allow this would complicate Mite's implementation on other processors.

3.5 The flags

There are four flags: `C` for carry, `Z` for zero, `N` for negative and `V` for overflow. These are set by arithmetic operations, and may be acted on by conditional branches (see section 4.4).

The use of flags varies widely between processors. The manner in which flags are set after each instruction differs, and some processors have a dedicated flags register, while others allow a general-purpose register to be nominated to contain the flags resulting from a particular instruction. It is easiest for Mite to specify that every instruction sets the flags, and only to allow conditional branches.

4 The instruction set

The instruction set looks similar to that of a typical RISC processor. All operands are registers.

4.1 Arithmetic and logical operations

The usual complement of operations is provided. Each takes a destination (or two in the case of `DIV`), which caters to three-operand machines, but is still easy to translate on two-operand machines, requiring just an extra register move instruction when the destination is different from the left-hand operand.

Twos-complement arithmetic is assumed; to allow other forms of negative numbers would add unnecessary complication, and the vast majority of current processors use twos-complement arithmetic.

`ADD`, `SUB` and <*MUL*> perform the corresponding single-length arithmetic operations. `DIV` is the only elaborate arithmetic instruction. It may produce either the quotient or the remainder or both, and may be signed or unsigned. Both floored and rounded-to-zero division are provided. Having a single instruction to perform division and remainder supports machines which perform division in software, as most division algorithms calculate the quotient and remainder simultaneously. Conversely, allowing either to be discarded supports machines with division hardware, as they often have separate instructions for division and remainder.

The logical operations `AND`, `OR` and `XOR` are provided, as well as logical shift left, and logical and arithmetic shift right, `SRL` and `SRA`. The more esoteric logical instructions provided by some processors were omitted along with bitwise rotates both to ease implementation and because they are rarely used by compilers.

Comparisons may be performed by omitting the destination of `SUB`, `AND` or `XOR`; the operation is performed but only the flags are affected.

4.2 Addressing memory

`LD_`w x, `[`a`]` loads the w bytes from the address in a, which must be a multiple of w, into x. `ST_`w x, `[`a`]` stores the least significant w bytes of x at the address in a. Other addressing modes must be emulated; a sophisticated implementation of Mite could recognize and generate some of them.

4.3 Code blocks

Code blocks divide a program into sections with a single entry point. A code block is of the form `c`<*label*> `[`<*reg*>`, ...] [ `<*directive*>`...]`. The label can be a branch destination and the list of registers in square brackets contains those registers active on entry to the block which are used during it. When a register is spilled the spill code can be placed outside the block if the register is not listed at the start; this mechanism pushes spill code out of loops where possible. Code blocks may be nested.

4.4 Branching

The instruction `B`*c* *a* performs a conditional branch to address *a* based on condition *c*. The condition is evaluated according to the current state of the flags. There are fourteen conditions covering all the usual tests and comparisons; the special conditional code `AL` (`AL`ways) causes an unconditional branch.

4.5 Subroutine call and return

Subroutine call and return is an area fraught with difficulty. It must be possible to implement very efficiently, while providing for procedure call standards, which differ widely from system to system, to be implemented transparently, so that Mite code can interoperate with native code.

The instruction `CALL` *a*, `[`p_1,...,p_n`]`, `[`r_1,...,r_m`]` calls the subroutine at *a* with parameters p_i and results r_j. Only the registers passed to the subroutine may be accessed by it (apart from any it creates), and return values must be passed in the r_j registers. Any registers not passed to or returned from the subroutine are preserved. This seemingly high-level instruction allows subroutines to be implemented with the normal low-level subroutine instructions, while enabling function call to be added with a small extension.

The subroutine return address is assumed to be stored in a register, and it can optionally be saved on the stack. This allows optimisation of leaf routines on machines which do not automatically save return addresses on the stack.

4.6 Other features

Other features which have been omitted for reasons of space include a general-purpose trap instruction which performs implementation-defined actions, and data blocks, which hold literal data and reserve space. Mite code is organized into modules which may be loaded and translated independently.

4.7 An example program

As a concrete example, here is a subroutine which calculates the factorial of its input:

```
cd.fact [1_t]
[
  NEW 2_t=1
  cd.loop [1_t,2_t]
  [
    NEW 3_tc=1
    MUL 2,2,1
    SUB 1,1,3
    BNE .loop
    KILL 3
  ]
  KILL 1
  RETR
]
```

5 Extensions

Several extensions to Mite have been considered; some have previously been part of its specification, but were considered too high-level or specialized to be included

in the core, and others are useful to support various sorts of system that one may want to build on top of Mite. In particular, an extension to support floating-point arithmetic is necessary if Mite is to be used for numerical work. An extension to support procedure call standards has already been designed to allow Mite to inter-operate with native code.

6 Current and future work

At the moment Mite is being implemented for the ARM and Intel x86 series of processors, along with a back end for the LCC C compiler to produce Mite code. The procedure calling standard extension will also be implemented, so that C programs can be compiled into Mite modules and then run either under Linux on the Intel or Acorn RISC OS on the ARM. Since no linker will be written the system will be restricted to single-file programs which call only the standard library.

Future research could extend in several directions:

Making Mite secure Access control policies are above the level of Mite; for Mite, security consists in ensuring that programs may not have undefined effects. This means addressing the following issues:

1. Out-of-range loading, storing and branching. This is the trickiest problem, as there are several ways of attacking it involving different trade-offs between flexibility and speed.
2. Using up memory through infinite recursion or allocating large stack frames. This is the same problem that confronts procedure call standard designers, and requires safe stack extension with well defined failure modes.

It would also be worth considering proof-carrying code, virtual code that carries with it a proof that it satisfies certain criteria.

Distribution and concurrency In order to support distributed computing, it might be worth building distribution primitives into Mite. It would be sensible to put them on a formal basis, such as a process calculus, which would allow for formal techniques to be applied to programs and protocols. This could be valuable given the difficulty of programming distributed system owing to concurrency and failure. The main problem would be to support the range of idioms currently in use for dealing with concurrency and distribution.

Formalizing Mite As the value of formal methods is increasingly being realised in building large software systems reliably, a formally defined and even verified virtual machine might be a great help. As Mite is small it should be relatively easy to specify formally, yet as it is fundamental to the systems built on it the returns would be disproportionately great.

Building Mite into an OS It should not be hard to port an operating system such as Linux to Mite. If the VM were built into the kernel, it would be possible to write most of the kernel and device drivers in virtual code. Linux programs would become binary-portable across all the platforms it supports, with a consequent leap in ease of use.

Investigating compilation for VMs Mite has been designed to work well with existing compiler technology; in the longer term it would seem more sensible to adapt compilers to VMs. This requires a reformulation of optimization in terms of machine-dependent and machine-independent components, and the

discovery of exactly what information can be usefully embedded in virtual code so as to produce optimizations easily when it is translated.

7 Mite and Forth

Apart from the fact that most Forth compilers have historically used a virtual machine, it is not obvious what Forth and Mite have in common, especially as Forth's stack-based nature seems diametrically opposed to Mite's use of registers. In fact, this difference is an illusion, and there is a transformation from the register version of Mite to a stack version, the only difference being that that the stack version lacks register usage information.

The instructions all become zero-operand, working on the top of the stack as in Forth, and instead of the register directives, stack operators are introduced. It is straightforward to translate mechanically between the two forms; to give a flavour of the result, here is the factorial program from section 4.7 in stack form:

```
cd.fact
[
  #1 PUSH
  cd.loop
  [
    1 PICK  MUL
    1 ROLL  #1 PUSH  SUB
    1 ROLL  BNE .loop
  ]
  1 ROLL  DROP  RET
]
```

The translation here is naïve: whenever a register is needed on top of the stack it is `ROLL`ed there if it is to be updated, and `PICK`ed otherwise. Constant registers are pushed whenever their value is needed. At the end of the loop, the stack order must be restored to the same as at the start, hence the final `1 ROLL`.

Note that translating from stack Mite to register Mite is much the same problem as compiling Forth for a register machine, and doing it efficiently is hard, as the information lost in translating from register to stack form must be reconstructed.

8 Conclusion

Virtual machines have enjoyed renewed interest in recent years as an aid to distributing executable content across the World Wide Web. To harness the full potential of distributed computing operating systems must be built on top of VMs, which must therefore be more flexible and efficient. Mite attempts to address these issues, while allowing specialization and extension for particular applications. Some correspondences to Forth were also noted. Focusing on the virtual machine as a processor abstraction layer should clarify many issues of design and usage and help hasten the arrival of ubiquitously portable and distributable software.

References

[1] Sun Microsystems Computer Corporation. *The Java Virtual Machine Specification*, 1995. Release 1.0 Beta.

[2] Phil Winterbottom and Rob Pike. The design of the Inferno virtual machine, 1997.