The design of the Inferno virtual machine

Phil Winterbottom and Rob Pike
Bell Labs, Lucent Technologies
`{philw,rob}@plan9.bell-labs.com`

Abstract

Virtual machines are an important component of modern portable environments such as Inferno and Java because they provide an architecture-independent representation of executable code. Their performance is critical to the success of such environments, but they are difficult to design well because they are subject to conflicting goals. On the one hand, they offer a way to hide the differences between instruction architectures; on the other, they must be implemented efficiently on a variety of underlying machines. A comparison of the engineering and evolution of the Inferno and Java virtual machines provides insight into the tradeoffs in their design and implementation. We argue that the design of virtual machines should be rooted in the nature of modern processors, not language interpreters, with an eye towards on-the-fly compilation rather than interpretation or special-purpose silicon.

Dis, the Inferno Virtual Machine

In early 1995, we set out to apply the ideas of the Plan 9 operating system [1] to a wider range of devices and networks. The resulting system, Inferno [2], is a small operating system and execution environment that supports application portability across a wide variety of processors and operating systems. Unaware of the contemporary work to establish Java [3] from the technology of the Oak project, we independently concluded that a virtual machine (VM) was a necessary component of such a system [4]. Because of improvements in processor speed and the feasibility of on-the-fly compilers, a VM can execute quickly enough to be economically viable.

The Inferno virtual machine, called Dis, has several unusual aspects to its design: the instruction set, the module system, and the garbage collector.

The Dis instruction set provides a close match to the architecture of existing processors. Instructions are of the form

```
OP    src1, src2, dst
```

The *src1* and *dst* operands specify general addresses or arbitrary-sized constants, while the *src2* operand is restricted to smaller constants and stack offsets to reduce code space. Each operand specifies an address either in the stack frame of the executing procedure or in the global data of its module.

The types of operands are set by the instructions. Basic types are `word` (32-bit signed), `big` (64-bit signed), `byte` (8-bit unsigned), `real` (64-bit IEEE floating point), and pointer (implementation-dependent). The instruction set follows the example of CISC processors, providing three-operand memory-to-memory operations for arithmetic, data motion, and so on. It also has instructions to allocate memory, to load modules, and to create, synchronize, and communicate between processes.

A module is the unit of dynamically loaded code and data. Modules are loaded by a VM instruction that returns a pointer to a method table for the module. That pointer is managed by the VM's garbage collector, so code and data for the module are garbage collected like any other memory. Type safety is preserved by checking method types at module load time using an MD5 signature of the type.

Memory management is intimately tied to the instruction set of the VM. Dis uses a hybrid garbage collection scheme: most garbage is collected by simple reference counting, while a real-time coloring collector gathers cyclic data. Because reference counting is an exact rather than conservative form of garbage collection, the type of all data items must be known to the VM run-time system. For this reason, the language-to-VM compiler generates a type descriptor for all compound types. This descriptor reports the location of all pointers within the type, allowing the VM to track references as items are copied.

Garbage collection

Memory dominates the cost of small systems, so the VM should be designed to keep memory usage as small as possible. Through reference-counted garbage collection, Dis reclaims memory the moment it becomes unused. Reference counting also eliminates the need for a large arena as required for efficient mark-and-sweep collection. Both these results reduce the memory requirements of the VM and its applications.

Compare this to the Java VM, whose instruction set makes it difficult to track references as objects are copied. This biases against reference counting, so JVM implementations choose lazier techniques such as mark-and-sweep, inducing a larger arena and delayed collection, both of which increase the memory use and therefore the cost of the overall system.

Issues in compiling

It is easy to interpret the individual instructions of a stack-based virtual machine (SM) such as the Java virtual machine (JVM), because most operands are implicit. However, a high-level language implementation of the interpreter generates more memory traffic than the equivalent set of instructions in a memory transfer machine (MM) such as Dis. Consider the code to execute

```
c = a + b;
```

An SM would execute this by a code burst such as this, which we have annotated with its memory traffic using L for load and S for store:

```
push  a      # LS
push  b      # LS
add          # LLS
store c      # LS
```

The corresponding MM code burst would be the plain three-operand instruction

```
add   a,b,c  # LLS
```

When interpreting, the extra memory traffic of the SM is masked by the time saved by not decoding any operand fields. The operand fields are implicit in the SM instructions, while the MM they are explicit: three operand fields must be decoded in every instruction, even those without operands.

When compiling, the tradeoffs are different. Clearly, either design can produce the same native instructions from its just-in-time compiler (JIT), but in the SM case most of the work must be done in the JIT, whereas in the MM design the front end has done most of the work and the JIT can be substantially simpler and faster.

A JIT for an SM is forced to do most of the work of register allocation in the JIT itself. Because the types of stack cells change as the program executes, the JIT must track their types as it compiles. In an MM, however, the architecture maps well to native instructions. This produces a continuum of register allocation strategies from none, to simple mapping of known cells to registers, to flow-based register allocation. Most of the work of any of these strategies can be done in the language-to-VM compiler. It can generate code for an infinite-register machine, and the JIT can then allocate as many as are available in the native architecture. Again, this distribution of work keeps the JIT simple.

Processors

The same issues that face the JIT writer also face the designer of special-purpose processors to support a VM. Register allocation in the JIT is analogous to register relabeling in silicon, and an SM design adds unnecessary complexity to an already difficult problem. One might argue that a stack-based processor design would mitigate the difficulties, but our experience with the implementation of a stack machine in the AT&T Crisp microprocessor [5] leads us to believe that stack architectures are inherently slower than register-based machines. Their design lengthens the critical path by replacing simple registers with a complex stack cache mechanism.

In other words, it is a better idea to match the design of the VM to the processor than the other way around.

Dis fits this criterion better, but we do not plan to implement Dis in silicon. The idea of a VM is to be architecture-independent; offering a special processor to run it negates the original goal by favoring one instruction set. Ignoring that for the moment, though, there could still be two reasons to consider designing silicon for Dis: performance and cost.

On performance, history shows that language-specific CPUs are not competitive. The investment in the special design takes energy away from the systems issues that ultimately dominate performance. Performance gains realized through language-specific support tend to be offset by parallel improvements in general-purpose processors during the life cycle of the CPU.

Dis compiles quickly into native code that runs only 30-50% slower than native C. At the current rate of processor improvement, that is only a few months of processor design time. It is wiser to focus on improving execution on commodity, general purpose processors than on inventing a new architecture.

The issue of cost is more subtle. Dis is close enough to familiar architectures that a special chip with high integration of systems facilities could be cost-effective on small platforms. The real reason for that, though, is that the memory management design of the virtual machine makes it easy to implement Dis in small memory. By contrast, whatever cost gains an integrated Java processor might realize will likely be lost in the extra memory required by its conservative garbage collection scheme [6].

References

1. R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. ‘‘Plan 9 from Bell Labs’’, *J. Computing Systems* 8:3, Summer 1995, pp. 221-254.

2. Dorward, S., et al., ‘‘Inferno’’, *IEEE Compcon 97 Proceedings*, 1997.

3. Arnold, K. and Gosling, J., *The Java Programming Language*, Addison-Wesley, 1996.

4. Nori, K. V., Ammann, U., Nabeli, H. H., and Jacobi, Ch., ‘‘Pascal P Implementation notes’’, in Barron, D. W. (ed.), *Pascal&emdash;The Language and its Implementation*, Wiley, 1981, pp. 125-170.

5. Ditzel, D. R. and McLellan, R., ‘‘Register Allocation for Free: The C Machine Stack Cache’’, *Proc. of Symp. on Arch. Supp. for Prog. Lang. and Op. Sys.*, March, 1982, pp. 48-56.

6. Case, B., ‘‘Implementing the Java Virtual Machine’’, *Microprocessor Report*, March 25, 1996, pp. 12-17.