The Design of the Inferno Virtual Machine

Phil Winterbottom
Rob Pike
Bell Labs, Lucent Technologies
{philw, rob}@plan9.bell-labs.com
http://www.lucent.com/inferno

Introduction

- Virtual Machine are topical
  - Intrinsically portable
  - More important because of networking
  - Should be fast enough for general use
- Conflicting Goals
  - Hide difference between architectures
  - Must run fast on general purpose machines
- Want VM to be competitive with compiled programming languages
- Claim: design for on-the-fly compilation not interpretation

Context

- Fundamental shift in Telecom industry toward data communications
- More diverse networks
  - LAN, Wireless, Fiber
  - IP, Ethernet, ATM
- Technology changing more quickly
  - hardware lifetime much shorter
- Need software systems that are portable, small, interoperate, based on network computing
- Current projects: Voice+IP router, firewall, ITS

Inferno

- Environment for portable network-centric applications
- Server and client architecture
- Limbo, Dis VM
- Virtual Operating system
  - same system interfaces and services everywhere
- Virtual network
  - same network interfaces and facilities everywhere
- Virtual graphics environment
  - same look and feel everywhere

Dis Instructions

- Memory-to-memory architecture
  - looks like a CISC cpu, not an abstract machine
- Three operand instructions
  - *OP src1, src2, dst*
    - src1, dst are general memory addresses
    - src2 restricted to constants and indirect addresses with small offsets
- All memory addresses are offsets from stack or module pointer
  - no absolute addresses
  - software memory protection

Dis Instructions

- Special instructions for processes, communications, etc.
- Pointers are explicit, and pointer cells store only valid addresses or *nil*
  - makes reference counting possible
  - (c.f. Java which puns cells, requiring runtime type tagging for r.c.

Garbage collection

- Desires:
  - small memory
  - constant, predictable overhead for real-time
  - fast collection
- No single GC can do this; Dis uses hybrid
  - Exact reference counting
    - instant free, bounded time, smallest footprint
  - RT incremental coloring garbage collector
    - recovers circular references, runs during idle time

Garbage collection

- Conservative mark and sweep requires more memory
  - typical for Java implementations
  - larger arena for efficient execution
  - larger high-water mark because of uncollected garbage
- GC algorithm selection is done during code generation in the language compiler

Interpretation

- Memory traffic depends on instruction set.

  Consider: `c = a+b`

- Stack machine (SM) implementation

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

- Memory-to-memory (MM)

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

Interpretation

- MM has less memory traffic, but costs are masked by need to decode operands.
- SM's implicit operands simplify instruction decode and reduce overhead of fetch execute

Compilation

- Tradeoffs change when using JIT compiler
  - Although JIT for SM or MM can produce the same code, where and when the work is done different
- Want to do all static analysis in front end (language to VM) compiler

Compilation

- Easier to approach this in MM:
  - Storage allocation done statically at compile time
  - no puns
- In SM:
  - Stack floats; cells change type
  - JIT must allocate storage and register to map onto native instructions
- These conditions dominate because
  - in production, will always use JIT
  - only interpret when debugging

Existing processors

- It is better to match the design of the VM to the processor than the other way around
- Existing processors are register based, not stack based
  - VM should emulate the predominant underlying architectures
- Stack machines are easy to interpret harder and more expensive to JIT

Special-purpose processors

- What about designing a special processor for VM
- Considerations are similar to designing JIT
  - register relabeling <=> register allocation
  - naïve stack machines produce more memory traffic
  - using stack caches to reduce traffic lengthens critical paths and cycle times
- So Dis would be a better starting point
  - but it's easily compiled so why bother?

Special-purpose processors

- In other words, JVM is hard to compile, so silicon looks attractive;
  - a better design would make silicon unnecessary
- Language-specific processors have never succeeded
  - They're always behind the technology curve
- Besides, special purpose silicon negates portability goal of a VM

Conclusion

- Better to design VM to match processor architectures than the other way around
- Still need more work to meet our goals
  - performance isn't as good as we'd like
  - register allocation needs to be done better
- Dis compiles quickly to native code that runs 30%-50% slower than native C.
  - Only a few months of processor design time
- Design toward on-the-fly compilation, not interpretation