Norman Ramsey

VM Design and Implementation

Steven Lucco
Microsoft Bay Area Research Center

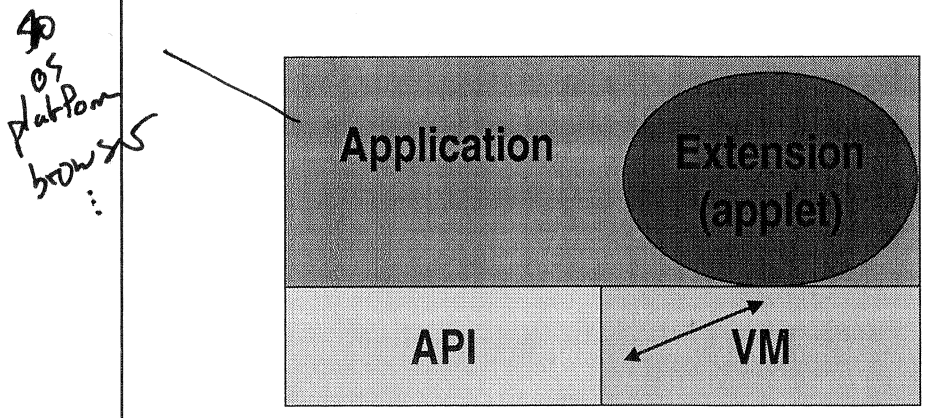
Today's agenda

- 1:30 greeting, introduction, participant interests
- 1:40 VM design trade-offs
- 2:20 ten-minute break
- 2:30 VM implementation

Virtual machine examples

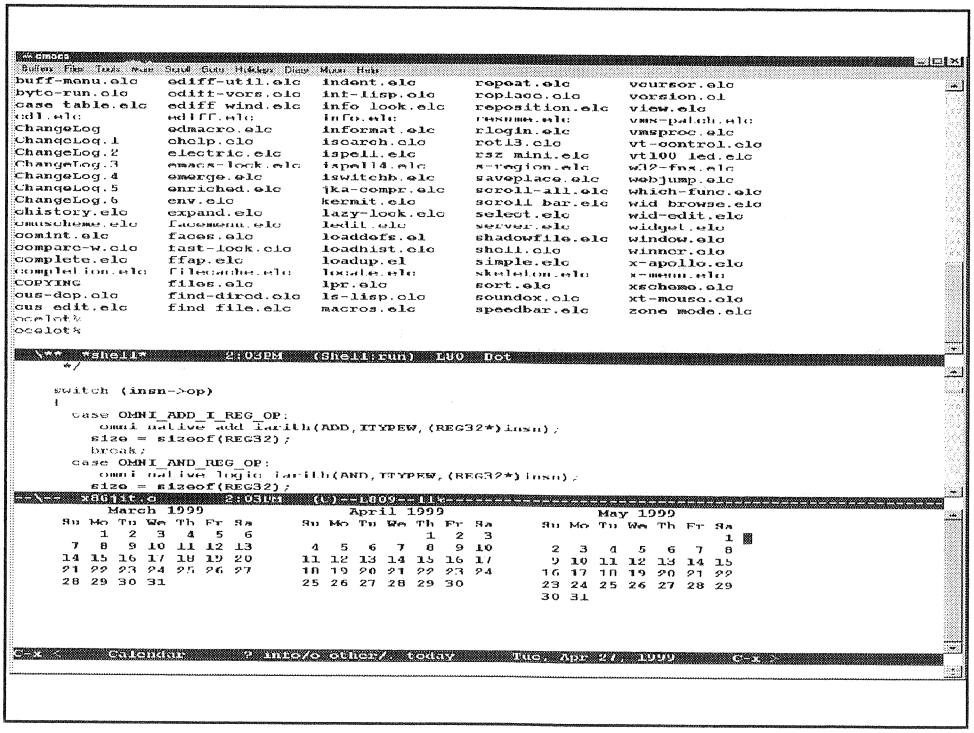
- Emacs and elisp
  - application-specific API
  - interpreted extension language
  - bytecoded for fast interpretation

Extensible Application VM



Emacs and elisp

- Have survived 20 years
- Clients diverse and powerful:
  - e-mail, browser, process control, calendar, binary editor, debug interface, help, font selection
- Elisp is “write once run everywhere” (x86, RISC, Unix, Windows)



Printer/document languages

- All HP printers are virtual machines for PCL
- Many printers virtual machines for Postscript
  - ghostscript, Adobe Acrobat reader also Postscript virtual machines
- Write once, run on many processors

JavaScript, VBScript replacing Java on Web sites ASCII only, like PostScript...

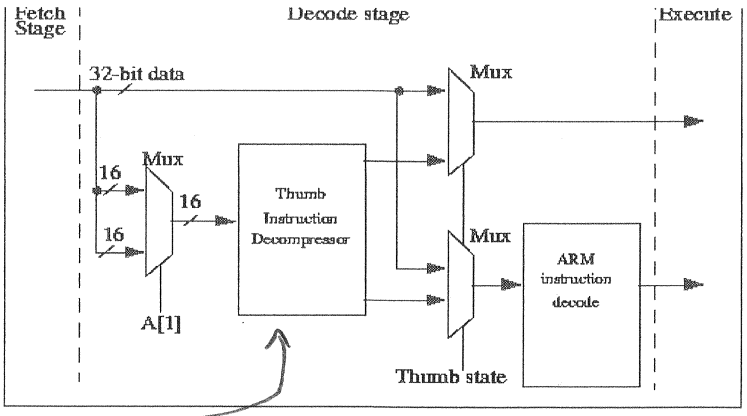
Postscript

- Unlike elisp, no bytecodes (interpreted ASCII script)
- Semantics include simple stack machine (VM semantics explicit)
- API tailored to bitmapped graphics

Hardware virtual machines

- ARM/Thumb
  - 16-bit instruction set embedded within 32-bit instruction set
  - interpreted by decode stage of pipeline
  - purpose is program compression

ARM pipeline (www.arm.com)



VM in one cycle!

More recently

- Browser as extensible application
- Java, Java APIs, Java bytecodes
- VBScript, JavaScript, Telescript
- Inferno
- TCL/Tk, Python, Perl, P-code
- Colusa VM (Omniware)

VM design features and goals

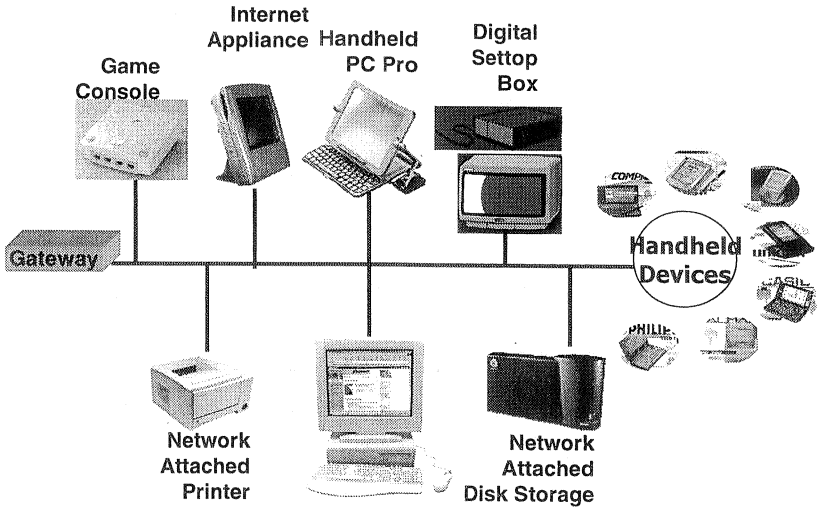
- Goals include:
  - security, encapsulation, portability, extensibility, small program size
- Will discuss goals and recent work in terms of specific features (code size, sandboxing, etc.)
- First describe deployment scenarios

VMs are proliferating

- Browsers (stock ticker)
- Servers (e-mail filter) — more memory OK, but must be fast
- Embedded systems (HTTP server for configuration)
- Set-top boxes (system software updates)

Mike Stonebraker: Ingres → RTI

"Those gizmos we'll all have..."



Safety

- = sandboxing + API security
- in practice, most problems occur in API structure and interface
  - inherent trade-off between API power and security (e.g. write file)
  - large "trusted computing base"

Sandboxing solutions

untrusted code in its own address space

- Address space
  - simple, clear model, but costly to cross protection boundary — IAC required!
- Software Fault Isolation (SFI)
  - lower crossing cost, but 4-20% run-time overhead for untrusted code; linearly verifiable

Sandboxing solutions (2)

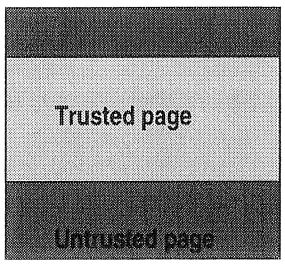
- Typesafe language
  - may reduce run-time overhead vs. SFI but restricts language choice
  - verification requires reading type description in addition to code
  - precise API definition
- Author authentication (no sandboxing)

fine granularity, requires, e.g., array bounds checks.

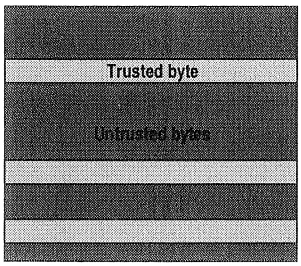
Verisign

Sandboxing granularity

- Memory granularity of API must match sandboxing granularity



Address space/SFI



Type safety

hard to merge things unless API is very coarse-grained

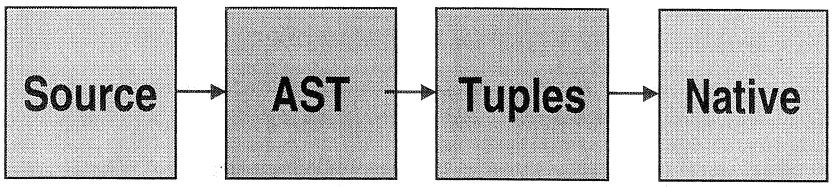
— e.g., might change style of API so trusted bytes won't leak out to untrusted code; e.g., use "handles" instead of pointers

Portability

- Processor-neutral program representation
- Interpretation or JIT compilation
- Format of static program data
  - little- or big-endian; floating point

so, need explicit bit-insert and bit-extract instructions in VM (to preserve "endian-neutrality")

Compilation stages



JavaScript,Post script, TCL, VBscript	Java, Pcode	Omniware	Standard program

Where to optimize (1)

- At client
  - requires memory proportional to largest procedure
  - inevitable trade-off between code quality (size, run time) and time needed to load application
  - enables late version binding and calling convention matching

Where to optimize (2)

- At application generation
  - Can spend time and memory on excellent optimization
  - Must include information to match calling conventions and for GC
  - Processor-specific optimization still done on client

Language support

- Using type system for sandboxing restricts language choice
  - TALx86 attempts to address this
- GC not always appropriate
- specialized instructions
  - invoke special

} 4-page description of its semantics

Program obscurity

- AST representation less obscure because reflects original syntax
  - e.g. Mocha decompiler for Java
- Optimized programs more obscure but it's a spectrum

Program size

- Download time, load time, RAM/ROM space, and working set all relevant
- For embedded systems, RAM/ROM space especially important
- Fast decompression required for acceptable app load and run time

Versions and binding

- Determine component structure at load time
  - enables late version substitution
  - limits benefit of pre-optimization
  - makes fast load difficult because of dynamic VM data structures
  - can alleviate by guessing bindings in advance

Performance

- Determines breadth of applications
- Several performance dimensions:
  - load time
  - download time
  - run time
  - code quality -> working set

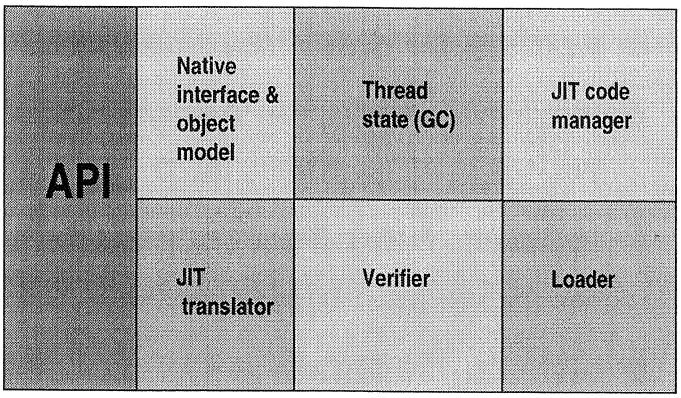
System integration

- Object model
- I/O
- Exceptions, threads
- Floating point
- Paging and sharing JIT code

Agenda for part two

- Safety
- Program Representation
- Program Compression
- Garbage Collection
- VM organization
- Moving forward

VM implementation



→ performance — how well is code statically laid out, so you can just map it, w/o calling malloc.

threads & re-entrancy!

(no interpreter)

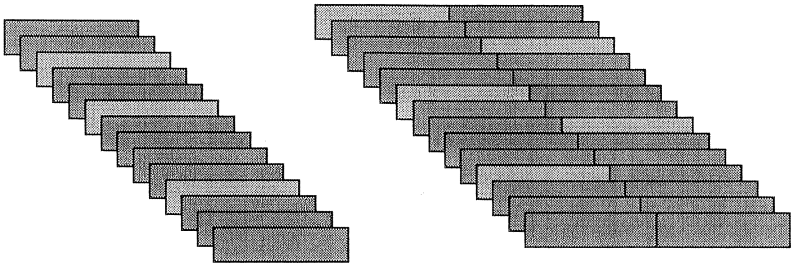
Sandboxing implementation (1)

- Type safety
  - verified by type flow analysis
  - works by ensuring self-consistency
  - requires run-time checks
    - array bounds, typecast
  - requires control over object allocation

Sandboxing implementation (2)

- Software Fault Isolation
  - verified by data flow analysis
  - works by ensuring consistency with page/segment map
  - requires run-time checks
    - indirect (jump, read, write)
  - can reduce run-time checks using type information and optimization

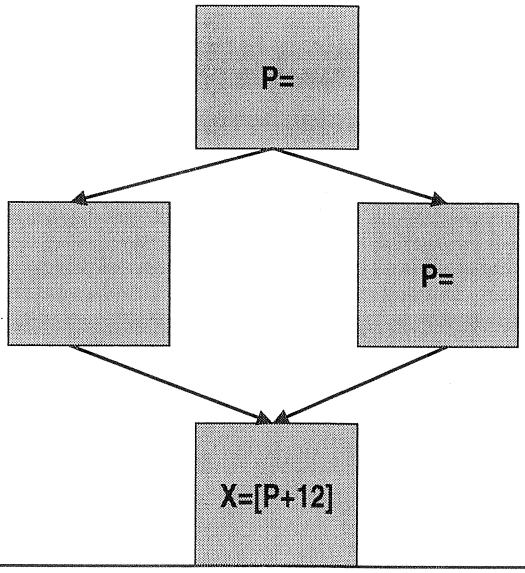
Adding meta-capabilities to programs



Single instruction stream

Multiple instruction stream

Def-use analysis



Program representation

- Is optimization possible?
  - Register allocation
- Can programs read/write program counter?
  - Continuations, setjmp, threads
- Stack or explicit registers?

— not with the Java VM
- requires big resources in JIT compiler to do register allocation

Stack code

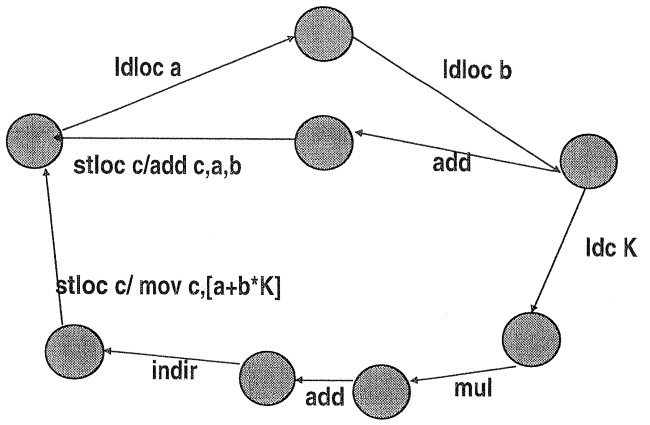
Amenable to very lean, simple VMs / translators

- c = [a+b*8]
  - ldloc a
  - ldloc b
  - ldc 8
  - mul
  - add
  - indir
  - stloc c

NOT true that stack code is inherently more compact than register code — what is true is you can trade code size for optimization

SIZE REDUCTION IN STACK CODE
- lots of unnamed temps on the stack
- specialized instructions to reduce code size

State machine translator



Abstract interpretation

- Dup, swap, require actual load time stack
- state machine recognizers significantly faster for stack code
  - can build addressing modes from simple instructions

don't have these sorts of instructions.

→ precludes stack-machine translator (input is no longer tree-like; LR DAG-like instead)

Register code

- c=[a+b*8]
  - mul b,b,8
  - add a,a,b
  - ld c,[a]
- with complex addressing modes
  - ld c,[a+b<<3]

Table-driven translator

VM	translation
add n0,n1,n2	lea eax,[edx+ecx]
beq n0,8,L1	cmp eax,8; jeq L1
mov n0 n1	mov eax, edx

Fastest translator (Colusa) — all optimization done in advance

BUT must commit to a # of registers? but see "virtual register sets" — Lucco, 1999?)

Calling conventions

- Parameter types required for native interface calls
  - complicates linking and program compression
- Caller- and callee-saved registers differ among target processors
- Function pointers require thunks

must communicate calling convention before doing good register allocation. "
but see "wacky"

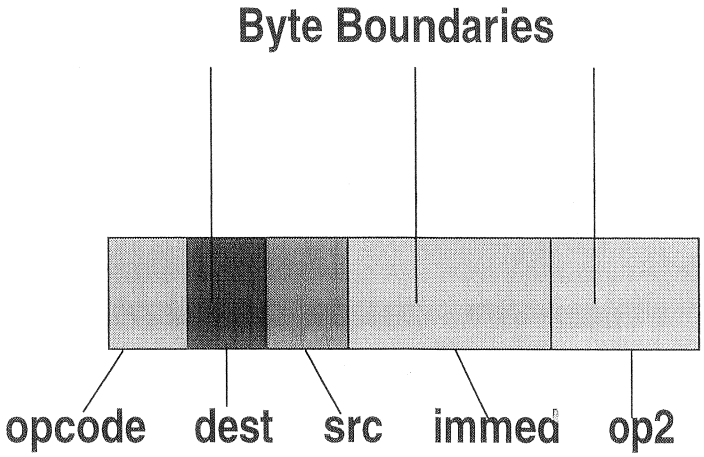
Program compression

- Wire compression
  - useful for download bottleneck
- Interpretable compression
  - useful for fast load and incremental JIT translation (run time)
- Current best numbers for both use *split-stream* compression

words = 25,000 Pmus need 6,000 to start up. Incremental JIT is important!

"semantic nuggets" don't fall on byte boundaries

Machine instructions



Multiple information streams

- View programs as multiple information streams
- Collect semantically related items into physically separate streams
- Consider relationships among streams

Wire format(1)

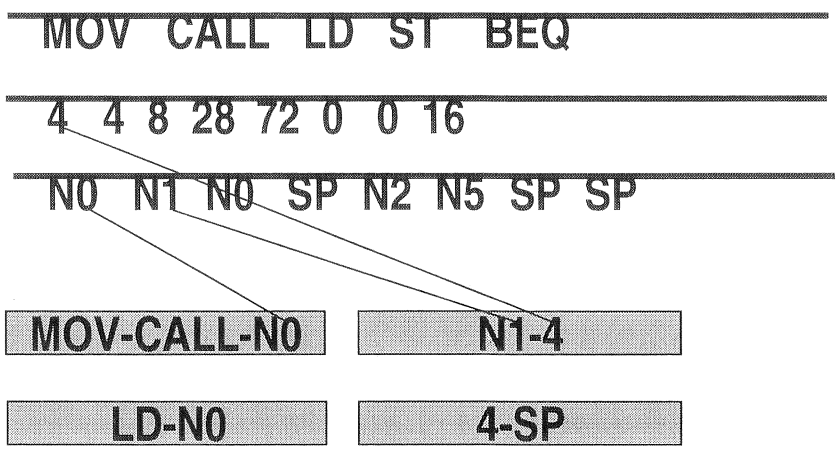
- Compile to trees
- Separate into streams
  - 1 with nested operators
  - 1 each for each type of operator that requires a literal operand
- Use move-to-front (MTF) coding on some streams

Wire format(2)

- Huffman code MTF indices
- Byte-pad MTF indices and tables
- Apply LZ compression (gzip)

MOV	CALL	LD	ST	BEQ
4	4	8	28	72	0	0	16
N0	N1	N0	N0	N2	N5	SP	SP

Byte-coded RISC (BRISC)



"Interpretable compression"

Garbage collection (1)

- Precise GC
  - offers potential cache benefits
  - requires root information for registers and stack slots
  - not appropriate for all languages
- Generational GC
  - requires write barriers
  - can be done conservatively

Be very careful about low choice of GC algorithm will affect implementation of compiler

GC table is 40% of size of code!

(to support precise GC at every instruction)

Garbage collection (2)

- Incremental is desirable for UI
- Precise+instruction granularity requires a lot of root information
  - proportional to size of program
- Interacts with object model
  - e.g. Java statement ("finalize" – making object live again considered ugly...)

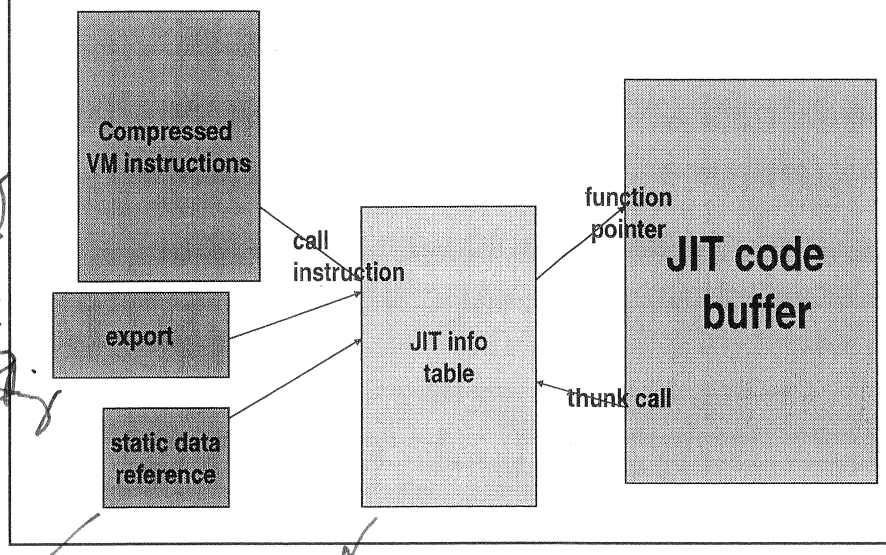
VM organization

- Intimately connected with VM program organization
  - link-time data structure setup yields faster load times
- Need linker-generated table to track incremental JIT translation
  - merge with object vtables

static layout for as much as possible

back patching beats indirect call even with cost of ~~updating~~ invalidating I-cache

JIT code management



function pointers

8 bytes/function

Moving forward

- Nested register-set allocation
  - compact representation for different register-set sizes
  - constant folding, CSE
- JIT code management in OS
  - share JIT code pages among processes; OS decompresses
  - working set tuning

Moving forward (2)

- More inclusive type systems
  - type-inferencing verification
  - lower-level representation (e.g. TALx86)
  - hybrid SFI and types
- Very simple virtual machines
  - such as state machines described earlier (for embedded systems)

Extra slides

Summary of Compression Numbers

- Wire format: compresses SPARC programs by factor of 5.3 and optimized x86 by 2.9 [PLDI99]
- BRISC (byte-coded RISC) is smaller than gzipped x86 by 20%
  - interpret at 12X native, JIT compile at 9M/sec, run at 6% overhead [PLDI97]

UCSD Pascal/P-code

- Stack machine bytecodes
- Microsoft Word 6.0 for MAC
  - 80% P-code/20% native PPC
  - P-code was not machine portable (PPC-specific)
- 2.2X program compression