Virtual Machine Design
Lecture 1: Overview and History

Antero Taivalsaari
August 2003
Welcome!

• Welcome to the Virtual Machine Design Seminar.
• TUT seminar 8109035.

First time a VM design course has been organized in Finland.
• Glad to see that there is so much interest in this topic!
Goals

- Introduce you to the world of virtual machine (VM) design.

- Explain the key technologies that are needed for building virtual machines, such as automatic memory management, interpretation, multithreading, and dynamic compilation.
Structure of the Seminar

- The seminar consists of two parts:
  - Lectures (6 or 7 lectures in total)
  - Student presentations

- Lectures will be held in room TB223 on Wednesdays, 10:15 – 11:45 am.
  - Lecture attendance is not required but recommended.

- To get the credits (2 ov), you must prepare and give a presentation on a selected topic related to virtual machine design.
  - Presentations will begin in November.
  - A list of suggested topics will be available later.
Lecture Schedule (Preliminary)

- Sep 10: History and overview of VM design
- Sep 17: Memory management
- Sep 24: Interpretation and execution
- Oct 1: Multithreading, synchronization and I/O
- Oct 8: Internals of the Java virtual machine
  (Oct 15: No lecture)
  (Oct 22: No lecture)
- Oct 29: High performance VMs (guest lecture)
- Nov 5: Student presentations begin
About the Lecturer

• Built virtual machines since the mid-1980s.
  – Main interests in the 1980s/early 1990s: Forth, Smalltalk, Self, other “dynamic” OO languages.

• In 1997, moved to California to work on Java virtual machines at Sun Microsystems.
  – Wrote the K Virtual Machine (KVM) at Sun Labs in 1998.
  – KVM became the starting point for Java 2 Micro Edition (J2ME), a popular version of the Java platform for mobile devices.

  – Led early J2ME standards activities (CLDC 1.0/1.1)
  – Co-author of the first Java Series book on J2ME.
Introduction
What is a Virtual Machine?

• A virtual machine (VM) is an “abstract” computing architecture or computational engine that is independent of any particular hardware or operating system.

• “Software machine” that runs on top of a real hardware platform and operating system.

• Allows the same programs to run “virtually” on any hardware for which a VM is available.
Characteristics of a Virtual Machine

• A virtual machine typically introduces its own instruction set that is used for executing programs.
  – This instruction set is independent of the architecture of the host operating system.

• A virtual machine usually also has its own memory system.
  – Access to the memory system of the host operating system is minimized.

• In general, access to the host operating system is often limited and controlled by the virtual machine's native function interface.
Typical High-Level Architecture

- Application
- Virtual Machine
- Operating System
- Hardware
Example: Components of a Java Virtual Machine (JVM)

<table>
<thead>
<tr>
<th>Class Loader and JAR Reader</th>
<th>Verifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreter and Execution Stacks</td>
<td>Compiler (optional)</td>
</tr>
<tr>
<td>Threading System and Thread Scheduler</td>
<td></td>
</tr>
<tr>
<td>Native Interface</td>
<td>Internal Runtime Structures</td>
</tr>
<tr>
<td>Memory System and Garbage Collector</td>
<td></td>
</tr>
</tbody>
</table>
Some Background

• Virtual machines have been studied and built since the late 1950s.

• Many early programming languages were built around the idea of having a portable runtime system.

• Yet VM design was always a fairly specialized topic; not many books or articles were written until recently.

• Popularity of the area exploded in the mid-1990s when the Java programming language was introduced.
Languages that Use Virtual Machines

• Well-known languages using a virtual machine:
  – *Basic*, 1964-1980s
  – *Forth*, early 1970s
  – *Pascal* (P-Code versions), late 1970s/early 1980s
  – *Smalltalk*, 1970s-1980s
  – *Self*, late 1980/early 1990s
  – *Java*, mid-1990s

• Numerous other languages:
  – ... *PostScript, TCL/TK, Perl, Python, C#, ...*
Why are Virtual Machines Interesting?

- Provide platform independence.
- Isolate programs from hardware details.
- Simplify application code migration.
- Can support dynamic downloading of software.
- Can provide additional security that machine-specific implementations cannot provide.
- Can hide complexity of legacy systems.

- Many programming languages are built around a virtual machine.
Virtual Machines vs. Operating Systems

• There is a lot of similarity between VM and OS design.
  – The key component areas are pretty much the same (memory management, multithreading, I/O, ...)

• A few key differences:
  – Virtual machines are usually designed to be as independent of the host operating system as possible.
  – Operating systems are “extensions of the underlying hardware”. They are built to facilitate access to the underlying computing architecture and maximize the utilization of the hardware resources.
  – Also, virtual machines are commonly tied to a particular programming language or language family.
  – Operating systems are usually language-independent.
Existing Material on VM Design

• There is a lot of material available on virtual machines.

• However, the material is scattered/fragmented and it is difficult to find any comprehensive presentations.

• A few books on the topic:

• Unfortunately, these books don't cover the area very well.
A Brief History of Programming Languages that Utilize a Virtual Machine
LISP

• John McCarthy, 1958
  – http://www-formal.stanford.edu/jmc/history/lisp/lisp.html
  – LISP is the second oldest programming language still in widespread use (after Fortran)

• LISP is characterized by the following ideas:
  – Computing with symbolic expressions rather than numbers,
  – representation of symbolic expressions and other information by list structure,
  – composition of functions as a tool for forming more complex functions out of a few primitive operations,
  – the representation of LISP programs as LISP data, and the function eval that serves both as a formal definition of the language and as an interpreter.
Sample Lisp Code

```
(define (primes)
  (letrec ((sieve ((lambda (s)
        (cons (car s)
        (delay (sieve (filter
                      (lambda (n)
                        (> (remainder n (car s)) 0))
                      (lambda (n)
                        (> (remainder n (car s)) 0))
                      (force (cdr s))))))))))
    (sieve (force (cdr nat)))))
```
Why is Lisp Interesting from VM Designer's Viewpoint?

• The first language to widely use garbage collection as a means of automating memory management.

• The first language to use recursion extensively.

• One of the first truly interactive languages that didn't require a “compile-link-execute-crash-debug” cycle.

• Lisp was one of the first systems where programs run in a “sandbox”; access to the operating system is limited and programs cannot really crash the system.

• The first truly “reflective” programming language as well; LISP has a very small language core; the rest of the system can be written in itself; programs can be manipulated as data.
UCSD Pascal

• The Pascal language was developed by Nicklaus Wirth in 1969.

• A fairly “conventional” programming language.
  – Predecessor to a large family of other languages (Modula..., Oberon...)

• Pascal did not become popular until Ken Bowles of the University of California San Diego (UCSD) implemented the *P-Code system* in the late 1970s.
  – A portable pseudocode system/language runtime.
  – The P-Code system made the implementation extremely portable, increasing the popularity of Pascal rapidly.
Sample Pascal Code

PROGRAM Fibonacci(input,output);

VAR
  lo : INTEGER; hi : INTEGER; n : INTEGER;
  golden_ratio : DOUBLE; ratio : DOUBLE;

BEGIN
  golden_ratio := (1.0 + sqrt(5.0))/2.0;
  lo := 1; hi := 1; n := 1;
  WHILE hi > 0 DO
    BEGIN
      n := n + 1; ratio := hi / lo;
      WRITELN(n : 2, ' ', hi, ratio : 25, ' ', (ratio - golden_ratio) : 21 : 18);
      hi := lo + hi; lo := hi - lo
    END
  END
Why is UCSD Pascal Interesting from VM Designer's Viewpoint?

• The P-Code system popularized the idea of using *pseudocode* to improve portability of programming language runtime systems.

• Uses a stack-oriented instruction set and five virtual registers.
  – Only one stack (no separate operand & call stacks)

• The first virtual machine implementation widely available to hobbyists.
  – Especially the Apple II implementation was very popular.
# P-Code Sample Instructions

<table>
<thead>
<tr>
<th>Inst.</th>
<th>Stack before</th>
<th>Stack after</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADI</td>
<td>i1 i2</td>
<td>i1+i2</td>
<td>add two integers</td>
</tr>
<tr>
<td>ADR</td>
<td>r1 r2</td>
<td>r1+r2</td>
<td>add two reals</td>
</tr>
<tr>
<td>DVI</td>
<td>i1 i2</td>
<td>i1/i2</td>
<td>integer division</td>
</tr>
<tr>
<td>INN</td>
<td>i1 s1</td>
<td>b1</td>
<td>set membership; b1 = whether i1 is a member of s1</td>
</tr>
<tr>
<td>LDCI</td>
<td>i1</td>
<td>i1</td>
<td>load integer constant</td>
</tr>
<tr>
<td>MOV</td>
<td>a1 a2</td>
<td></td>
<td>move</td>
</tr>
<tr>
<td>NOT</td>
<td>b1</td>
<td>~b1</td>
<td>boolean negation</td>
</tr>
</tbody>
</table>
BASIC

• Beginners All-purpose Symbolic Instruction Code.

• Developed by John Kemeny and Thomas Kurtz at Dartmouth College (USA) in mid-1960s.
  – http://www.kbasic.org/1/history.php3

• Interactive nature made it suitable for mini- and microcomputers (good timing!)

• Paul Allen and Bill Gates wrote the first interpreted implementation in 1975; this improved the portability of the language dramatically.
Sample BASIC Code

100 INPUT “Type a number”; N
120 IF N <= 0 GOTO 200
130 PRINT “Square root=” SQR(N)
140 GOTO 100
200 PRINT “Number must be > 0”
210 GOTO 100
Why is BASIC Interesting from VM Designer's Viewpoint?

• It really isn't very interesting...
  – The language has no specific contributions except ease of learning and ease of use.
  – Excessive use of GOTOs led to some horrible programs.

• However, the popularity of BASIC coincided with the microcomputer boom.
  – Many early microcomputer companies decided to integrate BASIC in their products.
  – You could either program in assembly language or BASIC...

• Some BASIC systems used pretty interesting intermediate code representation techniques.
Forth

- Invented by Charles Moore in the early 1970s.
  - http://www.forth.com/Content/History/History1.htm
- Originally designed to control radiotelescopes.
- Characteristics:
  - Forth is a “word-oriented” programming language; there is no syntax or grammar in the traditional sense.
  - All the primitive functions/words are also language keywords; open stack used for parameter passing.
  - Forth makes subroutine definition extremely cheap; this provides for extensibility and high level of procedural abstraction.
  - Extreme minimalism: The entire Forth system (including a simple multitasking programming environment) can fit in 8-15 kilobytes.
Sample Forth Code

: xReverse \ reverse the horizontal direction of the ball
   xStep @ +/- xStep ! ;
: yReverse \ reverse the vertical direction of the ball
   yStep @ +/- yStep ! ;
: checkLeft \ check for the left edge of the screen
   x @ 1 <= IF xReverse THEN ;
: checkRight \ check for the right edge of the screen
   x @ xMax >= IF xReverse THEN ;

ASCII o CONSTANT "ball"

: showBall \ draw the ball on the screen
   "ball" xyPlot ;
: hideBall \ hide (undraw) the ball
   "bl" xyPlot ;
: tryBall \ test the ball drawing routines
   BEGIN
   showBall
   checkLeft checkRight checkTop checkBottom
   hideBall xyStep
   AGAIN ;
Why is Forth Interesting from the VM Designer's Viewpoint?

• One of the easiest virtual machines to build.
• The VM consists of a small number of distinct components (stacks, dictionary, interpreter, virtual registers, primitives); no extra “fat”.
• The language itself is small, simple and efficient, and provides an unusual combination of high-level abstraction and very low level programming capabilities.
• High level of reflection (significant portions of the VM written in the language itself.)
• Ideal for embedded systems (if the awkward syntax is not exposed to the end user...)
Smalltalk

• Developed by Alan Kay's team at Xerox PARC
  – There are various versions (Smalltalk-72, -76, -80). Smalltalk-80 is the best known.
  – http://users.ipa.net/~dwighth/smalltalk/bluebook/bluebook_imp_toc.html

• Characteristics:
  – The first truly interactive object-oriented programming language (unlike Simula which was a compiler-based system.)
  – Took “everything is an object” and “message passing” metaphors to the extreme.
  – Everything is available for modification, even the VM itself (very high level of reflection.)
  – Even code is treated as objects (blocks).
  – The language is very closely coupled with a graphical interface; source code of a program cannot be easily separated from the programming environment.
Sample Smalltalk Code

| aString vowels |
aString := 'This is a string'.
vowels := aString select: [:aCharacter | aCharacter isVowel].

=================================================================

| rectangles aPoint |
rectangles := OrderedCollection
  with: (Rectangle left: 0 right: 10 top: 100 bottom: 200)
  with: (Rectangle left: 10 right: 10 top: 110 bottom: 210).
aPoint := Point x: 20 y: 20.
collisions := rectangles select: [:aRect | aRect containsPoint: aPoint].
Why is Smalltalk Interesting From the VM Designer's Viewpoint?

• Various implementation challenges:
  – Everything can be changed on the fly.
  – No static type system.
  – Even numbers are objects that are manipulated by message passing (= arithmetic operations can be slow.)
  – Blocks (heap-allocated code objects/stack frames) are difficult to implement efficiently.

• Extremely interactive/reflective => fun.

• Well-designed and mature class libraries => easy to write interesting software.

• There are high-quality public domain Smalltalk implementations available.
  – http://www.squeak.org/
Self

  - The majority of the actual implementation work was done at Sun Labs in the 1990s.
  - http://www.sunlabs.com/self

- Prototype-based flavor/variant of Smalltalk.
  - Took the “everything is an object” metaphor even further than Smalltalk.
  - No more classes; objects can inherit ("delegate") behavior directly from each other.
  - Extremely dynamic language: even the control structures or the inheritance relationships of objects can be changed on the fly.
acc: bankAccount copy.
acc balance: 100.
b: [acc deposit: 50].
acc balance.
"returns 100"
b value.
b value.
"returns 200"
acc balance.
Why is Self Interesting from VM Designer's Viewpoint?

• The Self language is so extremely dynamic that the implementors had to push the limits of VM technology very aggressively:
  – Adaptive compilation to speed up execution.
  – Generational garbage collection (originally invented by David Ungar in his Ph.D. work.)
  – Dynamic deoptimization to allow debugging of highly optimized programs.
  – Novel collaborative / visual programming and debugging environment (tightly integrated with the VM.)

• Many of the key technologies that are used today in mainstream Java virtual machines were invented by the Self group.
Java

- Developed by James Gosling's team at Sun Microsystems in the early 1990s.
  - [http://java.sun.com/people/jag/green](http://java.sun.com/people/jag/green)

- Originally designed for programming consumer devices (as a replacement of C++).
  - Uses a syntax that is familiar to C/C++ programmers.
  - Uses a portable virtual machine that provides automatic memory management and a simple stack-oriented instruction set.
  - *Class file verification* was added to enable downloading and execution of remote code securely.

- Again, great timing: the development of the Java technology coincided with the widespread adoption of web browsers in the mid-1990s.
Sample Java Code

class Peg {
    int pegNum;
    int disks[ ] = new int[64];
    int nDisks;

    public Peg(int n, int numDisks) {
        pegNum = n;
        for (int i = 0; i < numDisks; i++) {
            disks[i] = 0;
        }
        nDisks = 0;
    }

    public void addDisk(int diskNum) {
        disks[nDisks++] = diskNum;
    }

    public int removeDisk() {
        return disks[--nDisks];
    }
}
Why is Java Interesting from VM Designer's Viewpoint?

• Most people had never heard of virtual machines until Java came along!

• Combines a statically compiled programming language with a dynamic virtual machine.

• The Java virtual machine (JVM) is very well documented.

• A JVM is seemingly very easy to build.

• However, tight compatibility requirements make the actual implementation very challenging.
  – Must pass tens of thousands of test cases to prove compatibility.
Designing Virtual Machines
How are Virtual Machines Implemented?

- Virtual machines are typically written in "portable" and "efficient" programming languages such as C or C++.

- For performance-critical components, assembly language is used. The more machine code is used, the less portability.

- Some virtual machines (Lisp, Forth, Smalltalk) are largely written in the language itself. These systems have only a minimal core implemented in C or assembly language.

- Most Java VM implementations consist of a mixture of C/C++ and assembly code.
Typical High-Level Architecture

- Application
- Virtual Machine
- Operating System
- Hardware
Virtual Machine Design Considerations

- Size
- Portability
- Performance
- Memory consumption
- Scalability
- Security
- ...

There are always trade-offs in VM design!
Virtual Machine Design Considerations

• Unfortunately, for nearly all aspects of the VM:
  • Simple implies slow
  • Fast implies more complicated
  • Fast implies less portable
  • Fast implies larger memory consumption

  – Examples:
    • Interpretation
    • Memory management
    • Locking/Synchronization
    • Dynamic compilation
Components of a Virtual Machine

• The components of a virtual machine vary considerably depending on various factors:

  – Is the language interactive (Smalltalk, Forth) or non-interactive (Pascal, Java)?

  – Does the language have reflection capabilities (can you inspect or modify the VM or the program while it is running)?

  – Does the VM need to have performance that is comparable to non-interpreted systems?

  – Is multithreading support required?

  – Is the VM required to run in a “sandbox”?
Example: Components of a Java Virtual Machine (JVM)

<table>
<thead>
<tr>
<th>Class Loader and JAR Reader</th>
<th>Verifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreter and Execution Stacks</td>
<td>Compiler (optional)</td>
</tr>
<tr>
<td>Threading System and Thread Scheduler</td>
<td></td>
</tr>
<tr>
<td>Native Interface</td>
<td>Internal Runtime Structures</td>
</tr>
<tr>
<td>Memory System and Garbage Collector</td>
<td></td>
</tr>
</tbody>
</table>
Three Very Different Virtual Machines

• Ruka: A minimal, portable Forth virtual machine.

• KVM: A Java virtual machine implementation intended for small devices.

• Squeak: A feature-rich public domain Smalltalk implementation.
Ruka: A Portable Forth VM

- Written in ANSI C.
- 5,000 lines of code.
- Minimal executable size about 17 KB.

- Like all Forth systems, allows interactive definition and inspection of functions, and provides unrestricted access to the underlying operating system.

- Ported onto various small devices (Palm OS, PocketPC, SymbianOS, ...).
KVM: A Java VM for Small Devices

• Written in ANSI C.
• Version 1.0.4: about 35,000 lines of quite well-commented code.
  – About 50,000 lines if debugging support, native functions for the J2ME CLDC 1.0 libraries, and some network protocol primitives are added.
• Fully compliant with the J2ME CLDC test suite.
• Minimal executable size about 70 KB.
• Ported onto numerous commercial mobile phones all over the world.

http://wwws.sun.com/software/communitysource/j2me/cldc/download.html
Squeak: A Public Smalltalk VM

• A complete, compact implementation of the Smalltalk-80 Specification.
• Includes a very rich graphical programming environment and class library.
• The core VM is about 35,000 lines of C code.
• A complete executable with all the graphics libraries and plug-ins is about 1 MB.

• Various ports available.

http://sourceforge.net/projects/squeak/
More Information

• Virtual machine design:

• Designing “small memory” software:

• Conferences and workshops:
# Forthcoming Lectures

<table>
<thead>
<tr>
<th>Event</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Loader and JAR Reader</td>
<td>8.10</td>
</tr>
<tr>
<td>Interpreter and Execution</td>
<td>24.9</td>
</tr>
<tr>
<td>Threading System and Thread Scheduler</td>
<td>1.10</td>
</tr>
<tr>
<td>Native Interface</td>
<td>8.10</td>
</tr>
<tr>
<td>Verifier</td>
<td>29.10</td>
</tr>
<tr>
<td>Compiler (optional)</td>
<td></td>
</tr>
<tr>
<td>Internal Runtime Structures</td>
<td></td>
</tr>
<tr>
<td>Memory System and Garbage Collector</td>
<td>17.9</td>
</tr>
</tbody>
</table>
Questions?