9. Bytecode and Virtual Machines

Original material prepared by Adrian Lienhard and Marcus Denker
A virtual machine is an abstract computing architecture supporting a programming language in a hardware-independent fashion.
Roadmap

- Introduction
- Bytecode
- The heap store
- Interpreter
- Automatic memory management
- Threading System
- Optimizations
Implementing a Programming Language

- Pre-processor → Program
- Parser → Parse tree / IR
- Code Generator → Assembly code
- Assembler → Machine code
- Translator → Program
- Interpreter → Bytecode
- Bytecode Generator → Bytecode
- JIT Compiler → Bytecode Interpreter

© Oscar Nierstrasz
How are VMs implemented?

Typically using an *efficient and portable language* such as C, C++, or assembly code

Pharo VM platform-independent part written in *Slang*:
- subset of Smalltalk, translated to C
- core: 600 methods or 8k LOC in Slang
- Slang allows one to simulate VM in Smalltalk
Main Components of a VM

The heap store
Interpreter
Automatic memory management
Threading System
Pros and Cons of the VM Approach

Pros

> Platform independence of application code
  “Write once, run anywhere”
> Simpler programming model
> Security
> Optimizations for different hardware architectures

Cons

> Execution overhead
> Not suitable for system programming
Roadmap

> Introduction
> **Bytecode**
> > The heap store
> > Interpreter
> > Automatic memory management
> > Threading System
> > Optimizations
Reasons for working with Bytecode

> Generating Bytecode
  — Implementing compilers for other languages
  — Experimentation with new language features

> Parsing and Interpretation:
  — Analysis (e.g., self and super sends)
  — Decompilation (for systems without source)
  — Printing of bytecode
  — Interpretation: Debugger, Profiler
The Pharo Virtual Machine

> Virtual machine provides a virtual processor
  — Bytecode: The “machine-code” of the virtual machine

> Smalltalk (like Java): Stack machine
  — easy to implement interpreters for different processors
  — most hardware processors are register machines

> Pharo VM: Implemented in **Slang**
  — Slang: Subset of Smalltalk. (“C with Smalltalk Syntax”)
  — Translated to C
Bytecode in the CompiledMethod

> CompiledMethod format:

- **Header**: Number of temps, literals...
- **Literals**: Array of all Literal Objects
- **Bytecode**
- **Trailer**: Pointer to Source

(code snippets)
Bytecodes: Single or multibyte

> Different forms of bytecodes:
  
  — Single bytecodes:
    - Example: 120: push self

  — Groups of similar bytecodes
    - 16: push temp 1
    - 17: push temp 2
    - up to 31

  — Multibyte bytecodes
    - Problem: 4 bit offset may be too small
    - Solution: Use the following byte as offset
    - Example: Jumps need to encode large jump offsets

<table>
<thead>
<tr>
<th>Type</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>4 bits</td>
</tr>
</tbody>
</table>
Example: Number>>asInteger

> Smalltalk code:

```
Number>>asInteger
  "Answer an Integer nearest the receiver toward zero."
  ^self truncated
```

> Symbolic Bytecode

```
17 <70> self
18 <D0> send: truncated
19 <7C> returnTop
```
Example: Step by Step

> 17 <70> self
  — The receiver (self) is pushed on the stack

> 18 <D0> send: truncated
  — Bytecode 208: send literal selector 1
  — Get the selector from the first literal
  — start message lookup in the class of the object that is on top of the stack
  — result is pushed on the stack

> 19 <7C> returnTop
  — return the object on top of the stack to the calling method
Pharo Bytecode

> 256 Bytecodes, four groups:

— Stack Bytecodes
  - Stack manipulation: push / pop / dup

— Send Bytecodes
  - Invoke Methods

— Return Bytecodes
  - Return to caller

— Jump Bytecodes
  - Control flow inside a method
Stack Bytecodes

> Push values on the stack
  — e.g., temps, instVars, literals
  — e.g: 16 - 31: push instance variable

> Push Constants
  — False/True/Nil/1/0/2/-1

> Push self, thisContext

> Duplicate top of stack

> Pop
Sends and Returns

> Sends: receiver is on top of stack
  — Normal send
  — Super Sends
  — Hard-coded sends for efficiency, e.g. +, −

> Returns
  — Return top of stack to the sender
  — Return from a block
  — Special bytecodes for return self, nil, true, false (for efficiency)
Jump Bytecodes

> Control Flow inside one method
  — Used to implement control-flow efficiently
  — Example:

```
^ 1<2 ifTrue: ['true']
```

```
17 <76> pushConstant: 1
18 <77> pushConstant: 2
19 <B2> send: <
20 <99> jumpFalse: 23
21 <20> pushConstant: 'true'
22 <90> jumpTo: 24
23 <73> pushConstant: nil
24 <7C> returnTop
```
Roadmap

- Introduction
- Bytecode
- The heap store
- Interpreter
- Automatic memory management
- Threading System
- Optimizations
Object Memory Layout

32-bit direct-pointer scheme

Reality is more complex:
- 1-word header for instances of compact classes
- 2-word header for normal objects
- 3-word header for large objects
Different Object Formats

> fixed pointer fields

> indexable types:
– indexable pointer fields (e.g., Array)
– indexable weak pointer fields (e.g., WeakArray)
– indexable word fields (e.g., Bitmap)
– indexable byte fields (e.g., ByteString)

Object format (4bit)

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no fields</td>
</tr>
<tr>
<td>1</td>
<td>fixed fields only</td>
</tr>
<tr>
<td>2</td>
<td>indexable pointer fields only</td>
</tr>
<tr>
<td>3</td>
<td>both fixed and indexable pointer fields</td>
</tr>
<tr>
<td>4</td>
<td>both fixed and indexable weak fields</td>
</tr>
<tr>
<td>6</td>
<td>indexable word fields only</td>
</tr>
<tr>
<td>8-11</td>
<td>indexable byte fields only</td>
</tr>
<tr>
<td>12-15</td>
<td>...</td>
</tr>
</tbody>
</table>
Iterating Over All Objects in Memory

"Answer the first object on the heap"
apObject someObject

"Answer the next object on the heap"
apObject nextObject

Excludes small integers!

SystemNavigation>>allObjectsDo: aBlock
    | object endMarker |
    object := self someObject.
    endMarker := Object new.
    [endMarker == object]
      whileFalse: [aBlock value: object.
                    object := object nextObject]

|count|
count := 0.
SystemNavigation default allObjectsDo: [:anObject | count := count + 1].
Count

© Oscar Nierstrasz
Roadmap

- Introduction
- Bytecode
- The heap store
- **Interpreter**
- Automatic memory management
- Threading System
- Optimizations
Stack vs. Register VMs

VM provides a virtual processor that interprets bytecode instructions

Stack machines
- Smalltalk, Java and most other VMs
- Simple to implement for different hardware architectures
- Very compact code

Register machines
- Potentially faster than stack machines
- Only few register VMs, e.g., Parrot VM (Perl6)
Interpreter State and Loop

Interpreter state
- instruction pointer (ip): points to current bytecode
- stack pointer (sp): topmost item in the operand stack
- current active method or block context
- current active receiver and method

Interpreter loop
1. branch to appropriate bytecode routine
2. fetch next bytecode
3. increment instruction pointer
4. execute the bytecode routine
5. return to 1.
Method Contexts

- primitive index
- number of args
- number of temps
- large context flag
- number of literals
Stack Manipulating Bytecode Routine

Example: bytecode <70> self

Interpreter>>pushReceiverBytecode
    self fetchNextBytecode.
    self push: receiver

Interpreter>>push: anObject
    sp := sp + BytesPerWord.
    self longAt: sp put: anObject
Stack Manipulating Bytecode Routine

Example: bytecode <01> pushRcvr: 1

```smalltalk
Interpreter>>pushReceiverVariableBytecode
    self fetchNextBytecode.
    self pushReceiverVariable: (currentBytecode bitAnd: 16rF)

Interpreter>>pushReceiverVariable: fieldIndex
    self push: (self fetchPointer: fieldIndex ofObject: receiver)

Interpreter>>fetchPointer: fieldIndex ofObject: oop
    ^ self longAt: oop + BaseHeaderSize + (fieldIndex * BytesPerWord)
```
Message Sending Bytecode Routine

**Example:** bytecode `<E0> send: hello

1. find selector, receiver and its class
2. lookup message in the method dictionary of the class
3. if method not found, repeat this lookup in successive superclasses; if superclass is nil, instead send #doesNotUnderstand:
4. create a new method context and set it up
5. activate the context and start executing the instructions in the new method
Message Sending Bytecode Routine

**Example:** bytecode <E0> send: hello

Interpreter>>sendLiteralSelectorBytecode
  selector := self literal: (currentBytcode bitAnd: 16rF).
  argumentCount := ((currentBytecode >> 4) bitAnd: 3) - 1.
  rcvr := self stackValue: argumentCount.
  class := self fetchClassOf: rcvr.
  self findNewMethod.
  self executeNewMethod.
  self fetchNewBytecode

This routine (bytecodes 208-255) can use any of the first 16 literals and pass up to 2 arguments

\[<E0> = \text{E0} \text{ hex} = 224 \text{ dec} = 1110\ 0000 \text{ bin} \]

\[\text{E0 AND F} = 0\]
\[\Rightarrow \text{literal frame at 0}\]

\[((\text{E0} \gg 4) \text{ AND 3}) - 1 = 1\]
\[\Rightarrow 1 \text{ argument}\]
Primitive methods trigger a VM routine and are executed without a new method context unless they fail.

- Improve performance (arithmetics, at:, at:put:, ...)
- Do work that can only be done in VM (new object creation, process manipulation, become, ...)
- Interface with outside world (keyboard input, networking, ...)
- Interact with VM plugins (named primitives)
Roadmap

- Introduction
- Bytecode
- The heap store
- Interpreter
- **Automatic memory management**
- Threading System
- Optimizations
Automatic Memory Management

Tell when an object is no longer used and then recycle the memory

Challenges
- Fast allocation
- Fast program execution
- Small predictable pauses
- Scalable to large heaps
- Minimal space usage
Main Approaches

1. Reference Counting

2. Mark and Sweep
Reference Counting GC

**Idea**

- For each store operation increment count field in header of newly stored object
- Decrement if object is overwritten
- If count is 0, collect object and decrement the counter of each object it pointed to

**Problems**

- Run-time overhead of counting (particularly on stack)
- Inability to detect cycles (need additional GC technique)
Reference Counting GC
Mark and Sweep GC

Idea
> Suspend current process
> Mark phase: trace each accessible object leaving a mark in the object header (start at known root objects)
> Sweep phase: all objects with no mark are collected
> Remove all marks and resume current process

Problems
> Need to “stop the world”
> Slow for large heaps ➔ generational collectors
> Fragmentation ➔ compacting collectors
Mark and Sweep GC

![Diagram of Mark and Sweep GC process]

- **Root Set**: Objects that are reachable from the root set are marked.
- **Mark Phase**: Marks reachable objects as live.
- **Sweep Phase**: Cleans up unreachable objects.
Generational Collectors

**Idea**
- Partition objects into generations
- Create objects in young generation
- Tenuring: move live objects from young to old generation
- Incremental GC: frequently collect young generation (very fast)
- Full GC: infrequently collect young+old generation (slow)

**Difficulty**
- Need to track pointers from old to new space

*Most new objects live very short lives; most older objects live forever [Ungar 87]*
Generational Collectors: Remembered Set

Write barrier: remember objects with old-young pointers:

> On each store check whether stored object (object2) is young and storer (object1) is old

> If true, add storer to remembered set

> When marking young generation, use objects in remembered set as additional roots

```
object1.f := object2
```

© Oscar Nierstrasz
Compacting Collectors

**Idea**

> During the sweep phase all live objects are packed to the beginning of the heap

> Simplifies allocation since free space is in one contiguous block

**Challenge**

> Adjust all pointers of moved objects
  – object references on the heap
  – pointer variables of the interpreter!
The Pharo GC

Pharo: mark and sweep compacting collector with two generations

> Cooperative, i.e., not concurrent
> Single threaded
When Does the GC Run?

- Incremental GC on allocation count or memory needs
- Full GC on memory needs
- Tenure objects if survivor threshold exceeded

```
"Incremental GC after this many allocations"
SmalltalkImage current vmParameterAt: 5  4000

"Tenure when more than this many objects survive"
SmalltalkImage current vmParameterAt: 6  2000
```
### VM Memory Statistics

<table>
<thead>
<tr>
<th>Category</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory</td>
<td>20,245,028 bytes</td>
</tr>
<tr>
<td>old</td>
<td>14,784,388 bytes (73.0%)</td>
</tr>
<tr>
<td>young</td>
<td>117,724 bytes (0.6%)</td>
</tr>
<tr>
<td>used</td>
<td>14,902,112 bytes (73.6%)</td>
</tr>
<tr>
<td>free</td>
<td>5,342,916 bytes (26.4%)</td>
</tr>
<tr>
<td>GCs</td>
<td>975 (48ms between GCs)</td>
</tr>
<tr>
<td>full</td>
<td>0 totalling 0ms (0.0% uptime)</td>
</tr>
<tr>
<td>incr</td>
<td>975 totalling 267ms (1.0% uptime), avg 0.0ms</td>
</tr>
<tr>
<td>tenures</td>
<td>14 (avg 69 GCs/tenure)</td>
</tr>
<tr>
<td>Since last view</td>
<td></td>
</tr>
<tr>
<td>uptime</td>
<td>4.8s</td>
</tr>
<tr>
<td>full</td>
<td>0 totalling 0ms (0.0% uptime)</td>
</tr>
<tr>
<td>incr</td>
<td>90 totalling 29ms (1.0% uptime), avg 0.0ms</td>
</tr>
<tr>
<td>tenures</td>
<td>1 (avg 90 GCs/tenure)</td>
</tr>
</tbody>
</table>
“Force GC”
Smalltalk garbageCollectMost
Smalltalk garbageCollect

“Is object in remembered set, is it young?”
Smalltalk rootTable includes: anObject
Smalltalk isYoung: anObject

“Various settings and statistics”
SmalltalkImage current getVMPatterns

“Do an incremental GC after this many allocations”
SmalltalkImage current vmParameterAt: 5 put: 4000.

“Tenure when more than this many objects survive the GC”

“Grow/shrink heaproom”
SmalltalkImage current vmParameterAt: 25 put: 4*1024*1024.
SmalltalkImage current vmParameterAt: 24 put: 8*1024*1024.
Finding Memory Leaks

I have objects that do not get collected. What’s wrong?

– maybe object is just not GCed yet (force a full GC!)
– find the objects and then explore who references them

PointerFinder finds a path from a root to some object

```ruby
PointerFinder on:
  AssignmentNode someInstance

PointerExplorer new
  openExplorerFor:
    AssignmentNode
    someInstance
```
Roadmap

- Introduction
- Bytecode
- The heap store
- Interpreter
- Automatic memory management
- Threading System
- Optimizations
Multithreading is the ability to create concurrently running “processes”

Non-native threads (*green threads*)
- Only one native thread used by the VM
- Simpler to implement and easier to port

Native threads
- Using the native thread system provided by the OS
- Potentially higher performance
**Pharo: Green Threads**

Each process has its own execution stack, ip, sp, ...

There is always one (and only one) running process

Each process behaves as if it owns the entire VM

Each process can be interrupted (context switching)
Representing Processes and Run Queues

- ProcessorScheduler
  - activeProcess
  - processLists
- LinkedList
- Process
- suspendedContext
- MethodContext

Priority Levels:
- Processor: priority 80
- Processor: priority 70
- Processor: priority 10
Context Switching

Interpreter>>transferTo: newProcess

1. store the current ip and sp registers to the current context
2. store the current context in the old process’ suspendedContext
3. change Processor to point to newProcess
4. load ip and sp registers from new process’ suspendedContext

When you perform a context switch, which process should run next?
> *Cooperative* between processes of the same priority

> *Preemptive* between processes of different priorities

Context is switched to the first process with highest priority when:
- current process *waits* on a semaphore
- current process is *suspended* or *terminated*
- Processor *yield* is sent

Context is switched if the following process has a higher priority:
- process is *resumed* or created by another process
- process is *resumed* from a signaled semaphore

When a process is interrupted, it moves to the back of its run queue
Example: Semaphores and Scheduling

```smalltalk
to := false.
lock := Semaphore forMutualExclusion.
[lock critical: [to := true]] fork.
lock critical: [to assert: to not].
Processor yield.
to assert: to not.
Processor yield.
to assert: to
```

*When is the forked process activated?*
Roadmap

- Introduction
- Bytecode
- The heap store
- Interpreter
- Automatic memory management
- Threading System
- Optimizations
Many Optimizations...

> Method cache for faster lookup: receiver's class + method selector

> Method context cache (as much as 80% of objects created are context objects!)

> Interpreter loop: 256 way case statement to dispatch bytecodes

> Quick returns: methods that simply return a variable or known constant are compiled as a primitive method

> Small integers are tagged pointers: value is directly encoded in field references. Pointer is tagged with low-order bit equal to 1. The remaining 31 bit encode the signed integer value.

> ...

© Oscar Nierstrasz
Optimization: JIT (not in Pharo)

Idea: Just In Time Compilation
> Translate unit (method, loop, ...) into native machine code at runtime
> Store native code in a buffer on the heap

Challenges
> Run-time overhead of compilation
> Machine code takes a lot of space (4-8x compared to bytecode)
> Deoptimization (for debugging) is very tricky

Adaptive compilation: gather statistics to compile only units that are heavily used (hot spots)
References

> *Virtual Machines*, Iain D. Craig, Springer, 2006
> *Back to the Future – The Story of Squeak, A Practical Smalltalk Written in Itself*, Ingalls et al, OOPSLA ’97
> *Smalltalk-80, the Language and Its Implementation* (AKA “the Blue Book”), Goldberg, Robson, Addison-Wesley, ’83
> *Stacking them up: a Comparison of Virtual Machines*, Gough, IEEE’01
> *Virtual Machine Showdown: Stack Versus Registers*, Shi, Gregg, Beatty, Ertl, VEE’05
What you should know!

- What is the difference between the operand stack and the execution stack?
- How do bytecode routines and primitives differ?
- Why is the object format encoded in a complicated 4bit pattern instead of using regular boolean values?
- Why is the object address not suitable as a hash value?
- What happens if an object is only weakly referenced?
- Why is it hard to build a concurrent mark sweep GC?
- What does cooperative multithreading mean?
- How do you protect code from concurrent execution?
Can you answer these questions?

✎ There is a lot of similarity between VM and OS design. What are the common components?
✎ Why is accessing the 16th instance variable of an object more efficient than the 17th?
✎ Which disastrous situation could occur if a local C pointer variable exists when a new object is allocated?
✎ Why does #allObjectsDo: not include small integers?
✎ What is the largest possible small integer?
License

http://creativecommons.org/licenses/by-sa/3.0/

Attribution-ShareAlike 3.0 Unported

You are free:
  to Share — to copy, distribute and transmit the work
  to Remix — to adapt the work

Under the following conditions:
  Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
  Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

Any of the above conditions can be waived if you get permission from the copyright holder.

Nothing in this license impairs or restricts the author's moral rights.