VMs I Have Known and/or Loved

A subjective history

Mario Wolczko

Architect
Virtual Machine Research Group
Oracle Labs

http://labs.oracle.com
The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.
Overview

- Aims of this talk: entertain, educate, stimulate
- How? By talking about the VMs I’ve worked on/with, and what I learned from them:
  - **Self** (1993–’96)
  - JVMs: Exact VM, **HotSpot**, MaxineVM (1996–present)
  - **Truffle**, **Graal** and the **Alphabet Soup** (2010–)
Caveats

- Not a scholarly treatise - a personal view of the landscape out my window
- Mostly not my work but that of those around me
  - Credit where it’s due
  - Errors and omissions, are, of course, my own
- Intro and structure lifted from ICOOOLPS 2011 talk; one conclusion recanted
1984–1993

Smalltalk
Implementing Smalltalk Blue Book VM

Implementing Smalltalk-80 on the ICL PERQ

A dissertation submitted to the University of Manchester for the degree of Master of Science in the Faculty of Science.

October 1984

Mario I. Wolczko

Department of Computer Science
1984: Implementing the Smalltalk Blue Book VM

- Had never used or even seen a running Smalltalk system
1984: Implementing the Smalltalk Blue Book VM

- Had never used or even seen a running Smalltalk system
- Studied the “Blue Book” to learn Smalltalk, and understand the VM spec
1984: Implementing the Smalltalk Blue Book VM

- Had never used or even seen a running Smalltalk system
- Studied the “Blue Book” to learn Smalltalk, and understand the VM spec
- Implemented in ~10KLOC C
  - coding began July 2, first successful bring-up on Jan 14, 1985
1984: Implementing the Smalltalk Blue Book VM

- Had never used or even seen a running Smalltalk system
- Studied the “Blue Book” to learn Smalltalk, and understand the VM spec
- Implemented in ~10KLOC C
  - coding began July 2, first successful bring-up on Jan 14, 1985
- Demo
1984: Implementing the Smalltalk Blue Book VM

- Had never used or even seen a running Smalltalk system
- Studied the “Blue Book” to learn Smalltalk, and understand the VM spec
- Implemented in ~10KLOC C
  - coding began July 2, first successful bring-up on Jan 14, 1985
- Demo
- In retrospect:
  - A nice project for a student
  - My C code was awful
  - No better demonstration of Moore’s Law over 30 years
    - Perq 1 was inadequate (only 1MB RAM)
    - Used VAX 11/750+remote graphics (over RS-232!), Apollo and finally Perq 2
Smalltalk-80
An improvement over its successors
Smalltalk-80

An improvement over its successors

- Smalltalk-80 was an artifact from the future
  - Had been using paper tapes, teletypes and punched cards 3–5 years before
  - 9600 baud terminals were the norm; PCs and Macs had just appeared. The bitmapped display presented a megapixel at 30Hz
Smalltalk-80
An improvement over its successors

- Smalltalk-80 was an artifact from the future
  - Had been using paper tapes, teletypes and punched cards 3–5 years before
  - 9600baud terminals were the norm; PCs and Macs had just appeared. The bitmapped display presented a megapixel at 30Hz
- Demonstrated the power of virtualization
  - Implement a simple thing, get a complex thing
  - Virtual images transcend time and space
    - We see the same screen as someone at PARC on a spring morning in 1983
Smalltalk-80
An improvement over its successors

- Smalltalk-80 was an artifact from the future
  - Had been using paper tapes, teletypes and punched cards 3–5 years before
  - 9600baud terminals were the norm; PCs and Macs had just appeared. The bitmapped display presented a megapixel at 30Hz
- Demonstrated the power of virtualization
  - Implement a simple thing, get a complex thing
  - Virtual images transcend time and space
    - We see the same screen as someone at PARC on a spring morning in 1983
- “Meta-circular” definition — precise, concise
Main Lesson: Bytecode interpretation is slow

- Simple, but slow—2500 bytecode/s
- Even in microcode
  - Perq projected speed: 50kbps (6MHz CPU, 1.5MHz RAM)
  - Dorado: 400kbps@16MHz
Why so slow? 1. Bytecode dispatch

- Interpreter loop overhead; unpredictable branches on modern h/w

```java
for (;;) {
    BYTE b = getNextBytecode();
    switch (b) {
    case A: ...
    case B: ...
    ...
    }
}
```

- Various threading tricks can make it a few times faster
- But, still fundamentally inefficient
Why so slow?

- To execute: \( c = a + b \). “\( a=2, b=2 \)”

\[
b1 = \text{getNextBytecode}(); /* push a */ \\
\text{switch (b1) ...} \\
\text{fetch variable a and push onto stack} \\
b2 = \text{getNextBytecode}(); /* push b */ \\
\text{switch (b2) ...} \\
\text{fetch variable b and push onto stack} \\
b3 = \text{getNextBytecode}(); /* send + */ \\
\text{switch (b3) ...} \\
\text{send + to a with arg b} /* next slide */ \\
b4 = \text{getNextBytecode}(); /* pop and store into c */ \\
\text{switch (b4) ...} \\
\text{pop the top of stack and store in variable c}
\]
Why so slow? 2. Method dispatch

- Consider the execution of a simple expression, \( a+b \), in a dynamic language:
  - Find out the type of \( a \)
  - Find out the type of \( b \)
  - Find out what + means
  - Check that the operation is applicable to the data types, throw error if not
  - Prepare the data (e.g, strip tags)
  - Invoke the operation
  - Convert the result to canonical form (add tags)
C code for SmallInteger +

```c
SIGNED intArg, intRcvr;
int intRes;
OOP argOop= popStack;
if (isInt(argOop)) {
    intArg= intVal(argOop);
    OOP rcvrOop= popStack;
    if (isInt(rcvrOop)) {
        intRcvr= intVal(rcvrOop);
        intRes= intRcvr + intArg;
        if (isIntVal(intRes)) {
            NRpush(intObj(intRes));
            return FALSE;
        }
    }
}
unPop(2);
}
else
    unPop(1);
return TRUE;
```
C code for SmallInteger +

```c
SIGNED intArg, intRcvr;
int intRes;
OOP argOop= popStack;
if (isInt(argOop)) {
    intArg= intVal(argOop);
    OOP rcvrOop= popStack;
    if (isInt(rcvrOop)) {
        intRcvr= intVal(rcvrOop);
        intRes= intRcvr + intArg;
        if (isIntVal(intRes)) {
            NRpush(intObj(intRes));
            return FALSE;
        }
    }
}
unPop(2);
} else
    unPop(1);
return TRUE;

SIGNED intArg, intRcvr;
int intRes;
OOP argOop= *sp--;
if (argOop & 0x80000000) {
    intArg= argOop & 0x7fffffff;
    OOP rcvrOop= *sp;
    if (rcvrOop & 0x80000000) {
        intRcvr= rcvrOop & 0x7fffffff;
        intRes= intRcvr + intArg;
        if (intRes <= 0x3fffffff && intRes >= (-1<<30)) {
            *sp= intRes | 0x80000000;
            return FALSE;
        }
    }
} else
    sp += 2;
} else
    sp++;
return TRUE;
```
...Confirmed many times...

- “there was little hope for performance high enough to lure users away from traditional programming systems”
  
  – Joseph R. Falcone,  
  *The Analysis of the Smalltalk-80 System at Hewlett-Packard*
1986–1993: ParcPlace Smalltalk

- Landmark paper:
  - Appeared in POPL 1984
  - Major contributions:
    - **Just-in-time compilation** for an OO language
    - **Inlining caching** of method invocation targets
  - and:
    - Change of representation of contexts
    - Deutsch-Bobrow reference counting
Using ParcPlace Smalltalk

- I used—practically lived in—ParcPlace Smalltalk for ~5 years.
  - Sun 3/50, SPARCstation 1
- Rock-solid—I never encountered a VM bug
- Predictably performant
  - 20x faster than the Blue Book VM
  - Typically 5x slower than C code
  - But I found Smalltalk perhaps 10x more productive for my research
- Large increase in implementation complexity
  - Beyond a student project
  - First version in 68000 assembler, later in C
1993–1996

Self

The complexity of speed
The Self VM

- Self: like Smalltalk, only more so
- **Even harder** to make fast:
  - Variable accesses are via messages
  - Every control structure is implemented using blocks (closures)
  - Prototypes, not classes
  - Minimalist bytecode set
- Generational GC (Ungar)—problem solved?
- Craig Chambers’ compiler
  - Heroic efforts at optimization, but unpredictable
- Urs Hölzle’s compiler
  - Observation beats speculation
  - Count activations, observe messages and gather type info
  - Compile (or recompile) when you have a hot loop
  - Used the profile info to guide the compiler
  - Speculate that the past is a good predictor of the future
My Self experiences

- I joined in mid-’93—project ended 2 years later
- System was already fast
  - How fast? 1/3-1/2 C, sometimes faster (eg inlined recursive calls)
  - But had rough edges (GC, code quality, bugs)
  - Debugging via C++ debugger (gdb) was painful - wrong level of abstraction for many tasks
  - Careful use of C++ was a big improvement on C, even absent a C++ IDE
    - oop/map hierarchy -- OO in the VM
    - Duplicated functionality in different forms
      - E.g., GC barrier in C++ code, in emitted code (2 compilers)
1996–

Java VMs

From research to production
Java features that changed the game

- Primitive types—no tagging
- Built-in control flow, lack of closures—easier to compile
- Dynamic class loading, but not reflective program change
  - Later: Misha Dmitriev’s implementation of class redefinition
- Concurrency
- Awful bytecode design
- 1.0 VM: BlueBook-ish; conservative GC; “green” threads
- PEP - Java on Self (Agesen, with support from Ungar, me)
  - Demonstrated dynamic compilation and adaptive optimization for Java
  - Fast
- Considered—for a moment—converting the Self VM to Java
  - Didn’t know about HotSpot
1996–1999: The Exact VM

- Java 1.2 JVM for Solaris on SPARC and x86
- Derived from the “Classic” JVM (1.0, 1.1)
  - “Exactified” (Agesen, Detlefs)
- Initial goal was to provide PS-like performance, robustness for desktop and server workloads, for a limited lifetime (HotSpot acquisition in process)
  - Concurrency was important—Sun was selling lots of multiprocessor servers
  - Solaris thread support was good (!)
- Generational GC—but what about old space pauses?
- GC framework (Heller, White, Garthwaite, Flood)
  - Lots of GC research, leading to CMS, G1
  - Solaris threads were not so great after all — totally redone later by Roger Faulkner
- JIT compilation—awful bytecode design
- Later, basis of CVM (Sun’s J2ME CDC JVM: Kindle v1, BluRay)
1996–present: The HotSpot VM

- A start-up, Animorphic, founded in 1994 to build a high-performance Smalltalk VM starting from Self 3.0
  - Lars Bak and Urs Hölzle from the Self team, among others
- Neatly pivoted to Java
- Acquired by Sun
- Interpreter + compiler (rewritten later to become client compiler)
- Server compiler added
  - Much more sophisticated than predecessors
    - Click, Vick, Paleczny—the Rice compiler gurus—joined in 1997
- Still very much alive and in the lead
- Full-scale industrial development
  - Cast of >100 over the last 20 years?
2007–2013: The Maxine VM

- Started by Bernd Mathiske
- Influences from Klein (Self in Self), Jikes (Java in Java)
- Goal: make a fast but much more malleable VM
- Snippets—high-level description of intrinsic
- Inspector—VM-level abstractions for debugging and visualization
- Developed in Java IDE
- Compilers: CPS, C1X, Graal
2011–present

Dynamic Languages (again)
Recanting what I said at ICOOOLPS 2011

- VMs could be made fast, but at great effort and expense
- It didn’t look there were any big new ideas to be found, just lots of work to be done
- Boy, was I wrong
Relative speeds of various languages
(as measured by the Computer Language Benchmarks Game, ~1y ago)

slowdown
(smaller is better)

mean

C
C++
Java
JavaScript
Perl
PHP
Python
Ruby

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.
Building fast VMs is a lot of work

Speed

100x

10x

1x

Effort (person-years)

1m

1y

10y

100y

Simple JIT

HotSpot

Optimizing compiler

Inlining compiler

Self

Building fast VMs is a lot of work

Speed

100x

10x

1x

Effort (person-years)

1m

1y

10y

100y

Simple JIT

HotSpot

Optimizing compiler

Inlining compiler

Self
Building fast VMs is a lot of work

- **HotSpot**: Optimizing compiler
- **Self**: Inlining compiler
- **Simple JIT**: Most scripting languages

**Speed**
- 10x
- 1x

**Effort (person-years)**
- 1m
- 1y
- 10y
- 100y
- 100x
Building fast VMs is a lot of work

100x

10x

1x

Speed

Simple JIT

most scripting languages

HotSpot

Optimizing compiler

Inlining compiler

Self

Speed

Optimizing compiler

Inlining compiler

Simple JIT

most scripting languages
Building fast VMs is a lot of work

- **Speed**
  - 10x
  - 100x

- **KLOC**
  - 10
  - 100
  - 1000

- **Inlining compiler**
- **Self**
- **HotSpot** Optimizing compiler
- **Simple JIT**
- most scripting languages
The language designer’s dilemma

Current situation

Prototype a new language
- Parser and language work to build syntax tree (AST)
- Execute using AST interpreter

Write a “real” VM
- In C/C++
- Still using AST interpreter
- Spend a lot of time implementing runtime system, GC, …

People start using it

People complain about performance
- Define a bytecode format and write bytecode interpreter

Performance is still bad
- Write a JIT compiler
- Improve the garbage collector

Massive adoption
- Hire a big team of implementors to build an optimizing VM
The language designer’s dilemma

Current situation

Prototype a new language
Parser and language work to build syntax tree (AST)
Execute using AST interpreter

Write a “real” VM
In C/C++
Still using AST interpreter
Spend a lot of time implementing runtime system, GC, …

People start using it

People complain about performance
Define a bytecode format and write bytecode interpreter

Performance is still bad
Write a JIT compiler
Improve the garbage collector

Massive adoption
Hire a big team of implementors to build an optimizing VM

Most give up here
Why not reuse a compiler from another language?

- Glue on a different front end
  - PEP, Smalltalk in Self, Smalltalk-to-Eiffel, Smalltalk-to-Common Lisp
- Many languages translate to Java bytecode
  - But don’t seem to go much faster
  - Is Java bytecode the problem? Too Java-specific.
- Compiler already has to interoperate with garbage collection
  - Conservative GC is unacceptable
  - Static compiler are usually too slow to be used dynamically
  - Retrofitting GC to an optimizing compiler is usually unsuccessful
    - Large rewrites necessary to preserve info
- Even if this worked well, it’s still a lot of hard work
A new approach: Truffle and Graal
Partial evaluation of self-specializing abstract syntax trees

Self-specializing interpreter nodes gather type and profile information

Optimizing compiler uses type, profile and AST structure to selectively inline and optimize

Conceived by Thomas Würthinger in 2011

Implementation by students and Oracle staff at Johannes Kepler University, Linz
Specializing interpreter nodes for the common case

VAL eval() {
    L = left.eval(); R = right.eval();
    switch (type(L)) {
        case INT: res = L + R;
            if (overflowed(L, R, res)) {...}
            else return BOX(res);
        case ....
    }
}
Specializing interpreter nodes for the common case

VAL eval() {
    L = left.eval(); R = right.eval();
    switch (type(L)) {
        case INT: res = L + R;
            if (overflowed(L, R, res)) {...}
            else return BOX(res);
        case ....
    }
    eval() eval()
}

int evalInt() throws Unexpected {
    try {
        return left.evalInt() + right.evalInt();
    } catch (Unexpected u) {
        // revert to slow version
    }
}

int+ int+

eval() eval() evalInt() evalInt()
Optimizing compilation driven by specialized ASTs

```java
int evalInt() throws Unexpected {
    try {
        return left.evalInt()+right.evalInt();
    } catch (Unexpected u) {
        // revert to slow version
    }
}
```

evalInt() {
    return val("a");
}

evalInt() {
    return val("b");
}
int evalInt() throws Unexpected {
    try {
        return left.evalInt()+right.evalInt();
    } catch (Unexpected u) {
        // revert to slow version
    }
}

int+ inline

int evalInt() {
    return val("a");
}

int evalInt() {
    return val("b");
}

evalInt() {
    return val("a")+val("b");
}
int evalInt() throws Unexpected {
    try {
        return left.evalInt()+right.evalInt();
    } catch (Unexpected u) {
        // revert to slow version
    }
}

inline

evalInt() {
    return val("a") + val("b");
}

compile

add Ra, Rb, Result
System architecture

Truffle Framework

JVM

your language here?
System architecture

Truffle Framework

Graal Compiler

HotSpot JVM

your language here?
System architecture

Truffle Framework

Graal Compiler

Standalone Substrate VM or HotSpot JVM

your language here?
System architecture

Truffle Framework

Graal Compiler

Standalone Substrate VM

or

HotSpot JVM

or

Substrate VM embedded

your language here?
System architecture

Truffle Framework

Graal Compiler

- Standalone Substrate VM
- HotSpot JVM
- Substrate VM embedded

GPU backend for Graal

Your language here?
The language designer’s dilemma—resolved?

Current situation

Prototype a new language
Parser and language work to build syntax tree (AST)
Execute using AST interpreter

Write a “real” VM
In C/C++
Still using AST interpreter
Spend a lot of time implementing runtime system, GC, …

People start using it

People complain about performance
Define a bytecode format and write bytecode interpreter

Performance is still bad
Write a JIT compiler
Improve the garbage collector

Massive adoption
Hire a big team of implementors to build an optimizing VM
The language designer’s dilemma—resolved?

Current situation

- **Prototype a new language**
  - Parser and language work to build syntax tree (AST)
  - Execute using AST interpreter

- **Write a “real” VM**
  - In C/C++
  - Still using AST interpreter
  - Spend a lot of time implementing runtime system, GC, …

- **People start using it**

- **People complain about performance**
  - Define a bytecode format and write bytecode interpreter

- **Performance is still bad**
  - Write a JIT compiler
  - Improve the garbage collector

- **Massive adoption**
  - Hire a big team of implementors to build an optimizing VM

How it should be

- **Prototype a new language in Java**
  - Parser and language work to build syntax tree (AST)
  - Execute using AST interpreter and optimizing compiler

- **People start using it**
  - And it is already fast
Tag elimination for dynamically-typed languages

- Chambers and Ungar came up with method customization for Self
  - Don’t have to inherit machine code for a machine; can customize and optimize for local behavior (e.g., variable overrides abstract method)
- Idea: customized methods for objects whose fields contain primitives (int, float)
  - Due to Thomas Würthinger (2010)
  - Eliminates need for tagging
    - although could be a halfway between unboxed and boxed representations
Relative speeds of various languages
(as measured by the Computer Language Benchmarks Game, ~1y ago)
Relative speeds of various languages
(as measured by the Computer Language Benchmarks Game, ~1y ago)

Goal:
Peak Performance: Truffle/JavaScript versus V8

Benchmarks from Octane v.1 suite, Hardware: Intel Core i7-3770, 16 GB RAM, V8 version 3.22.1 from 25-Sep-2013, Truffle/JavaScript: Running on Graal/OpenJDK changeset 63c378b7c1c3 from 26-Oct-2013
Peak Performance: Truffle/Ruby versus JRuby 1.7.5
Evidence that it can be done with modest effort
Slide from Chris Seaton, JVM Lang Summit 2013, describing his Ruby implementation on Truffle

Simplicity

- One intern working for five months on the Ruby implementation
- New to Truffle, Graal and Ruby
- Written using Eclipse
- Debugged as a normal Java program using the server compiler
- Run using Graal for testing and performance numbers
- No mention in the implementation of bytecode, classloaders, assembly, system calls, OSR
- One very minor use of Unsafe, one very minor use of reflection
Summary: reuse our stack, get a fast VM without a ton of work

Speed

100x

10x

1x

Effort (person-years)

1m

1y

10y

100y

HotSpot
Optimizing compiler

Self
Inlining compiler

Simple JIT
most scripting languages
Summary: reuse our stack, get a fast VM without a ton of work
For more information

*An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler*, Mon@4.00, "Regency B" (VMIL workshop)

How the Graal IR is good for optimizing Java code

*ZipPy on Truffle: A Fast and Simple Implementation of Python*, Demo, Wed@11.15, “Vision” room

A Truffle deep dive

*One VM to Rule Them All — Onward!* paper, Thu @ 10.30, Cosmopolitan B

Full paper on Truffle: http://dx.doi.org/10.1145/2509578.2509581

*So you want to be an industrial researcher?* SPLASH-I, Tue @ 1pm

http://openjdk.java.net/projects/graal/
https://wiki.openjdk.java.net/display/Graal/Publications+and+Presentations
graal-dev@openjdk.java.net
Acknowledgments

**Oracle Labs**
Laurent Daynès
Erik Eckstein
Michael Haupt
Peter Kessler
Christos Kotselidis
David Leibs
Roland Schatz
Doug Simon
Michael Van De Vanter
Christian Wimmer
Christian Wirth
Mario Wolczko
Thomas Würthinger
Laura Hill (Manager)

**JKU Linz**
Prof. Hanspeter Mössenböck
Gilles Duboscq
Matthias Grimmer
Christian Häubl
Josef Haider
Christian Humer
Christian Huber
Manuel Rigger
Lukas Stadler
Bernhard Urban
Andreas Wöß

**Purdue University**
Prof. Jan Vitek
Tomas Kalibera
Petra Maj
Lei Zhao

**University of California, Irvine**
Prof. Michael Franz
Codrut Stancu
Gulfem Savrun Yeniceri
Wei Zhang

**University of Manchester**
Chris Seaton

**University of Edinburgh**
Christophe Dubach
Juan José Fumero Alfonso
Toomas Remmelg
Ranjeet Singh

**LaBRI**
Floréal Morandat

**T. U. Dortmund**
Prof. Peter Marwedel
Ingo Korb
Helena Kotthaus

**University of California, Davis**
Prof. Duncan Temple Lang
Nicholas Ulle

**Interns**
Danilo Ansaloni
Daniele Bonetta
Shams Imam
Stephen Kell
Gregor Richards
Rifat Shariyar
Hardware and Software

Engineered to Work Together