Advanced Design and Implementation of Virtual Machines

Xiao-Feng Li
Contents

Foreword, xvii
Preface, xix
About This Book, xxi
Author, xxiii

Section I  Basics of Virtual Machines

Chapter 1  Introduction of the Virtual Machine 3
  1.1  TYPES OF VIRTUAL MACHINES 3
  1.2  WHY VIRTUAL MACHINE? 4
  1.3  VIRTUAL MACHINE EXAMPLES 5
      1.3.1  JavaScript Engine 6
      1.3.2  Perl Engine 6
      1.3.3  Android Java VM 7
      1.3.4  Apache Harmony 8

Chapter 2  Inside of a Virtual Machine 9
  2.1  CORE COMPONENTS OF VIRTUAL MACHINE 9
      2.1.1  Loader and Dynamic Linker 9
      2.1.2  Execution Engine 10
      2.1.3  Memory Manager 10
      2.1.4  Thread Scheduler 11
      2.1.5  Language Extension 12
      2.1.6  Traditional Model versus Virtual Machine Model 13
  2.2  VIRTUAL ISA 14
      2.2.1  Java Virtual Machine 15
      2.2.2  JVM versus CLR 18
CHAPTER 3  • Data Structures in a Virtual Machine 21

3.1 OBJECT AND CLASS 21
3.2 OBJECT REPRESENTATION 22
3.3 METHOD DESCRIPTION 23

SECTION II  Design of Virtual Machines

CHAPTER 4  • Design of Execution Engine 27

4.1 INTERPRETER 27
  4.1.1 Super Instruction 28
  4.1.2 Selective Inlining 28
4.2 JIT COMPILATION 29
  4.2.1 Method-Based JIT 29
  4.2.2 Trace-Based JIT 32
  4.2.3 Region-Based JIT 35
4.3 RELATION BETWEEN INTERPRETER AND JIT COMPILER 36
4.4 AHEAD-OF-TIME COMPILATION 38
4.5 COMPILE-TIME VERSUS RUNTIME 40

CHAPTER 5  • Design of Garbage Collection 45

5.1 OBJECT LIFETIME 45
5.2 REFERENCE COUNTING 46
5.3 OBJECT TRACING 48
5.4 RC VERSUS OBJECT TRACING 50
5.5 GC SAFE POINT 51
5.6 COMMON TRACING GC ALGORITHMS 54
  5.6.1 Mark Sweep 54
  5.6.2 Trace Copy 55
5.7 VARIANTS OF COMMON TRACING GCs 57
  5.7.1 Mark-Compact 57
  5.7.2 Slide-Compact 57
  5.7.3 Trace Forward 58
5.7.4  Mark-Copy 58
5.7.5  Generational Collection 59
   5.7.5.1  Remember Set and Write-Barrier 60
   5.7.5.2  Cart-Table and Remember-Set Enumeration 61
5.8  MOVING-GC VERSUS NONMOVING GC 62
   5.8.1  Data Locality 62
   5.8.2  Bump-Pointer Allocation 62
   5.8.3  Free-List and Allocation Bitmap 63
   5.8.4  Size-Segregated List 63
   5.8.5  Mark Bits and Allocation Bits 64
   5.8.6  Thread-Local Allocation 64
   5.8.7  Hybrid of Moving and Nonmoving GC 66

CHAPTER 6  ■  Design of Threading 69
6.1  WHAT IS A THREAD 69
6.2  KERNEL THREAD AND USER THREAD 71
6.3  MAPPING OF VM THREAD TO OS THREAD 73
6.4  SYNCHRONIZATION CONSTRUCTS 75
6.5  MONITOR 77
   6.5.1  Mutual Exclusion 77
   6.5.2  Conditional Variable 78
   6.5.3  Monitorenter 78
   6.5.4  Monitorexit 81
   6.5.5  Object.wait() 83
   6.5.6  Object.notify() 84
6.6  ATOMICS 85
6.7  MONITOR VERSUS ATOMICS 88
   6.7.1  Blocking versus Nonblocking 88
   6.7.2  Central Control Point 88
   6.7.3  Lock versus No-Lock 88
   6.7.4  Blocking on Top of Nonblocking 89
### Contents

6.8 COLLECTOR AND MUTATOR 90  
6.9 THREAD-LOCAL DATA 92  
  6.9.1 Thread-Local Allocator 93  
6.10 THREAD SUSPENSION SUPPORT FOR GC 95  
  6.10.1 GC Safe Point 95  
  6.10.2 GC Safe Region 97  
  6.10.3 Lock-Based Safe Point 100  
  6.10.4 Thread Interaction in a Collection 101

**Section III** Supports in Virtual Machine

**Chapter 7** Native Interface 107  
  7.1 WHY NATIVE INTERFACE 107  
  7.2 TRANSITION FROM MANAGED CODE TO NATIVE CODE 109  
    7.2.1 Wrapper for Native Method 109  
    7.2.2 Wrapper for GC Support 112  
    7.2.3 Wrapper for Synchronization Support 113  
  7.3 BINDING OF NATIVE METHOD IMPLEMENTATION 114  
  7.4 TRANSITION FROM NATIVE CODE TO MANAGED CODE 115  
  7.5 TRANSITION FROM NATIVE CODE TO NATIVE CODE 118  
    7.5.1 Native-to-Native through JNI API 119  
      7.5.1.1 Native-to-Java Transition 120  
      7.5.1.2 Java-to-Native Transition 120  
    7.5.2 Why JNI API Is Used in Native-to-Native 121

**Chapter 8** Stack Unwinding 125  
  8.1 WHY STACK UNWINDING 125  
  8.2 STACK UNWINDING FOR JAVA METHOD FRAMES 126  
    8.2.1 Stack-Unwinding Design 126  
    8.2.2 Stack-Unwinding Implementation 129  
  8.3 STACK UNWINDING WITH NATIVE METHOD FRAMES 130  
    8.3.1 Stack-Unwinding Design 130  
    8.3.2 Java-to-Native Wrapper Design 132  
    8.3.3 Stack-Unwinding Implementation 135  
    8.3.4 Native Frame versus C Frame 137
# Chapter 9: Garbage Collection Support

## 9.1 WHY GC SUPPORT

## 9.2 SUPPORT GARBAGE COLLECTION IN JAVA CODE

- **9.2.1 GC-Map**
  - 9.2.1.1 *Runtime Update*
  - 9.2.1.2 *Compile-Time Generation*
  - 9.2.1.3 *Lazy Generation*
- **9.2.2 Stack-Unwinding with Registers**

## 9.3 SUPPORT GARBAGE COLLECTION IN THE NATIVE CODE

- **9.3.1 Object Reference Access**
- **9.3.2 Object Handle Implementation**
- **9.3.3 GC-Safety Property Maintenance**
- **9.3.4 Object Body Access**
- **9.3.5 Object Allocation**

## 9.4 SUPPORT GARBAGE COLLECTION IN A SYNCHRONIZED METHOD

- **9.4.1 Synchronized Java Method**
- **9.4.2 Synchronized Native Method**

## 9.5 GC SUPPORT IN TRANSITIONS BETWEEN JAVA AND NATIVE CODES

- **9.5.1 Native-to-Java**
- **9.5.2 Java-to-Native**
- **9.5.3 Native-to-Native**

## 9.6 GLOBAL ROOT-SET

# Chapter 10: Runtime-Helpers

## 10.1 WHY RUNTIME-HELPERS

## 10.2 VM-SERVICE DESIGN WITH RUNTIME-HELPERS

- **10.2.1 Operations of Runtime-Helpers**
- **10.2.2 Runtime-Helper Implementation**
- **10.2.3 JNI API as Runtime-Helper**
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.3</td>
<td>VM-SERVICE DESIGN WITHOUT RUNTIME-HELPER</td>
<td>176</td>
</tr>
<tr>
<td>10.3.1</td>
<td>Fast-Path of Runtime-Helpers</td>
<td>178</td>
</tr>
<tr>
<td>10.3.2</td>
<td>Programming for Fast-Path VM-Services</td>
<td>179</td>
</tr>
<tr>
<td>10.4</td>
<td>TYPICAL VM-SERVICES</td>
<td>180</td>
</tr>
<tr>
<td>11.1</td>
<td>SAVE CONTEXT OF EXCEPTION-THROWING</td>
<td>186</td>
</tr>
<tr>
<td>11.1.1</td>
<td>VM-Saved Context</td>
<td>186</td>
</tr>
<tr>
<td>11.1.2</td>
<td>OS-Saved Context in Linux</td>
<td>187</td>
</tr>
<tr>
<td>11.1.3</td>
<td>OS-Saved Context in Windows</td>
<td>188</td>
</tr>
<tr>
<td>11.1.4</td>
<td>Synchronous versus Asynchronous Exception</td>
<td>189</td>
</tr>
<tr>
<td>11.2</td>
<td>EXCEPTION HANDLING IN AND ACROSS THE NATIVE CODE</td>
<td>190</td>
</tr>
<tr>
<td>11.2.1</td>
<td>Exception Handling in the Native Code</td>
<td>190</td>
</tr>
<tr>
<td>11.2.2</td>
<td>Java Code with Exception Returns to the Native Code</td>
<td>191</td>
</tr>
<tr>
<td>11.2.3</td>
<td>Native Code with Exception Returns to the Java Code</td>
<td>196</td>
</tr>
<tr>
<td>11.3</td>
<td>SAVE STACK TRACE</td>
<td>197</td>
</tr>
<tr>
<td>11.4</td>
<td>FIND THE EXCEPTION HANDLER</td>
<td>199</td>
</tr>
<tr>
<td>11.5</td>
<td>TRANSFER THE CONTROL</td>
<td>202</td>
</tr>
<tr>
<td>11.5.1</td>
<td>Operations of Control-Transfer</td>
<td>202</td>
</tr>
<tr>
<td>11.5.2</td>
<td>Registers for Control Transfer</td>
<td>204</td>
</tr>
<tr>
<td>11.5.3</td>
<td>Data Register Restoration</td>
<td>205</td>
</tr>
<tr>
<td>11.5.3.1</td>
<td>Abrupt Completion of the Java Method</td>
<td>205</td>
</tr>
<tr>
<td>11.5.3.2</td>
<td>Control Transfer to the Exception Handler</td>
<td>206</td>
</tr>
<tr>
<td>11.5.4</td>
<td>Control-Register Fixing</td>
<td>207</td>
</tr>
<tr>
<td>11.5.5</td>
<td>Resume the Execution</td>
<td>207</td>
</tr>
<tr>
<td>11.5.5.1</td>
<td>Resume for Proactive Exception</td>
<td>207</td>
</tr>
<tr>
<td>11.5.5.2</td>
<td>Resume for Hardware-Fault Exception</td>
<td>209</td>
</tr>
<tr>
<td>11.5.6</td>
<td>Uncaught Exception</td>
<td>210</td>
</tr>
<tr>
<td>12.1</td>
<td>FINALIZATION</td>
<td>213</td>
</tr>
<tr>
<td>12.2</td>
<td>WHY WEAK REFERENCES</td>
<td>215</td>
</tr>
</tbody>
</table>
12.3 OBJECT LIFE-TIME STATES
  12.3.1 Object State Transition
  12.3.2 Reference Queue
  12.3.3 Reference-Object State Transition
12.4 REFERENCE-OBJECT IMPLEMENTATION
12.5 REFERENCE-OBJECT PROCESSING ORDER

CHAPTER 13 • Modularity Design of VM
  13.1 VM COMPONENTS
  13.2 OBJECT INFORMATION EXPOSURE
  13.3 GARBAGE COLLECTOR INTERFACE
  13.4 EXECUTION ENGINE INTERFACE
  13.5 CROSS-COMPONENT OPTIMIZATIONS

SECTION IV • Optimizations of Garbage Collection

CHAPTER 14 • Optimizing GC for Throughput
  14.1 ADAPTATION BETWEEN PARTIAL AND FULL-HEAP COLLECTIONS
  14.2 ADAPTATION BETWEEN GENERATIONAL AND NONGENERATIONAL ALGORITHMS
  14.3 ADAPTATION OF SPACE SIZE IN HEAP
    14.3.1 Space Size Extension
    14.3.2 NOS Size
    14.3.3 Partial-Forward NOS Design
    14.3.4 Semi-Space NOS Design
    14.3.5 Aged-Mature NOS Design
    14.3.6 Fallback Collection
  14.4 ADAPTATION BETWEEN ALLOCATION SPACES
  14.5 LARGE OS PAGE AND PREFETCH

CHAPTER 15 • Optimizing GC for Scalability
  15.1 COLLECTION PHASES
  15.2 PARALLEL OBJECT GRAPH TRAVERSAL
    15.2.1 Task Sharing
    15.2.2 Work-Stealing
    15.2.3 Task-Pushing
<table>
<thead>
<tr>
<th>Chapter 15</th>
<th>Parallel Compaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.3 PARALLEL MARKING OF OBJECTS</td>
<td>277</td>
</tr>
<tr>
<td>15.4 PARALLEL COMPACTION</td>
<td>279</td>
</tr>
<tr>
<td>15.4.1 Parallel LISP2 Compactor</td>
<td>279</td>
</tr>
<tr>
<td>15.4.2 Object Dependence Tree</td>
<td>280</td>
</tr>
<tr>
<td>15.4.3 Compactor with Target Table for Forwarding Pointer</td>
<td>284</td>
</tr>
<tr>
<td>15.4.4 Compactor with Section of Objects</td>
<td>286</td>
</tr>
<tr>
<td>15.4.5 In-Place Compactor in Single Pass</td>
<td>287</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 16</th>
<th>Optimizing GC for Responsiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1 REGIONAL GC</td>
<td>292</td>
</tr>
<tr>
<td>16.2 CONCURRENT TRACING</td>
<td>294</td>
</tr>
<tr>
<td>16.2.1 Snapshot-at-the-Beginning</td>
<td>295</td>
</tr>
<tr>
<td>16.2.1.1 Slot-Based SATB</td>
<td>296</td>
</tr>
<tr>
<td>16.2.1.2 Object-Based SATB</td>
<td>298</td>
</tr>
<tr>
<td>16.2.1.3 SATB Discussions</td>
<td>299</td>
</tr>
<tr>
<td>16.2.2 Incremental-Update</td>
<td>299</td>
</tr>
<tr>
<td>16.2.2.1 INC by Remember Reference</td>
<td>300</td>
</tr>
<tr>
<td>16.2.2.2 Second-Round Tracing for INC</td>
<td>300</td>
</tr>
<tr>
<td>16.2.2.3 INC by Remember Root</td>
<td>301</td>
</tr>
<tr>
<td>16.2.2.4 INC Discussions</td>
<td>302</td>
</tr>
<tr>
<td>16.2.3 Concurrent Tracing in Tricolor Terminology</td>
<td>303</td>
</tr>
<tr>
<td>16.2.4 Concurrent Tracing with Read-Barrier</td>
<td>304</td>
</tr>
<tr>
<td>16.3 CONCURRENT ROOT-SET ENUMERATION</td>
<td>305</td>
</tr>
<tr>
<td>16.3.1 Concurrent Root-Set Enumeration Design</td>
<td>306</td>
</tr>
<tr>
<td>16.3.2 Trace Heap during Root-Set Enumeration</td>
<td>309</td>
</tr>
<tr>
<td>16.3.3 Concurrent Stack Scanning</td>
<td>311</td>
</tr>
<tr>
<td>16.4 CONCURRENT COLLECTION SCHEDULING</td>
<td>312</td>
</tr>
<tr>
<td>16.4.1 Schedule Concurrent Root-Set Enumeration</td>
<td>312</td>
</tr>
<tr>
<td>16.4.2 Schedule Concurrent Heap Tracing</td>
<td>313</td>
</tr>
<tr>
<td>16.4.3 Concurrent Collection Scheduling</td>
<td>317</td>
</tr>
<tr>
<td>16.4.4 Concurrent Collection Phase Transitions</td>
<td>318</td>
</tr>
</tbody>
</table>
CHAPTER 17  • Concurrent Moving Collection 323

17.1 CONCURRENT COPYING: “TO-SPACE INVARIANT” 323
  17.1.1 Slot-Based “To-Space Invariant” 324
    17.1.1.1 Flipping Phase of “To-Space Invariant” 324
    17.1.1.2 Copying Phase of “To-Space Invariant” 324
  17.1.2 “To-Space Invariant” Properties 327
  17.1.3 Object Forwarding 329
  17.1.4 Object-Based “To-Space Invariant” 330
  17.1.5 Virtual Memory-Based “To-Space Invariant” 332

17.2 CONCURRENT COPYING: “CURRENT-COPY INVARIANT” 334
  17.2.1 Object-Moving Storm 334
  17.2.2 “Current-Copy Invariant” Design 334
  17.2.3 Concurrent Copying versus Concurrent Heap Tracing 337
    17.2.3.1 Concurrent Copying Based on Concurrent Tracing Algorithm 338
    17.2.3.2 Correct Design of “Current-Copy Invariant” 339

17.3 CONCURRENT COPYING: “FROM-SPACE INVARIANT” 340
  17.3.1 “From-Space Invariant” Design 340
    17.3.1.1 Write-Barrier for “From-Space Invariant” 340
    17.3.1.2 Heap Tracing for “From-Space Invariant” 341
  17.3.2 Partial-Forward “From-Space Invariant” 343

17.4 FULLY CONCURRENT MOVING WITHOUT STW 344

17.5 CONCURRENT COMPACTING COLLECTION 344
  17.5.1 Concurrent Regional-Copying Collection 344
    17.5.1.1 Single-Pass Regional Copying 344
    17.5.1.2 Separate Pass for Heap Tracing 345
    17.5.1.3 The Pass for Reference-Fixing 347
  17.5.2 Virtual Memory-Based Concurrent Compacting 348
    17.5.2.1 Fault Handler with Read-Barrier 348
    17.5.2.2 Fault Handler without Read-Barrier 350
    17.5.2.3 Virtual Semi-Space Implementation 352
    17.5.2.4 Concurrent In-Place Compaction 353
### Section V  Optimizations of Thread Interactions

#### Chapter 18  Optimizing Monitor Performance

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.1</td>
<td>LAZY LOCK</td>
<td>359</td>
</tr>
<tr>
<td>18.2</td>
<td>THIN-LOCK</td>
<td>361</td>
</tr>
<tr>
<td>18.2.1</td>
<td>Locking Path of Thin-Lock</td>
<td>361</td>
</tr>
<tr>
<td>18.2.2</td>
<td>Unlocking Path of Thin-Lock</td>
<td>365</td>
</tr>
<tr>
<td>18.2.3</td>
<td>Support Contention Flag Resetting</td>
<td>368</td>
</tr>
<tr>
<td>18.3</td>
<td>FAT-LOCK</td>
<td>370</td>
</tr>
<tr>
<td>18.3.1</td>
<td>Consolidated Monitor Data Structure</td>
<td>370</td>
</tr>
<tr>
<td>18.3.2</td>
<td>Offload Supports to OS</td>
<td>372</td>
</tr>
<tr>
<td>18.3.3</td>
<td>Thin-Lock Inflation to Fat-Lock</td>
<td>374</td>
</tr>
<tr>
<td>18.3.4</td>
<td>Sleep-Waiting for the Contended Thin-Lock</td>
<td>377</td>
</tr>
<tr>
<td>18.4</td>
<td>TASUKI LOCK</td>
<td>381</td>
</tr>
<tr>
<td>18.4.1</td>
<td>Use Same Fat-Lock Monitor for Contention Control</td>
<td>381</td>
</tr>
<tr>
<td>18.4.1.1</td>
<td>Access to Monitor</td>
<td>381</td>
</tr>
<tr>
<td>18.4.1.2</td>
<td>Inflation Process</td>
<td>381</td>
</tr>
<tr>
<td>18.4.1.3</td>
<td>Dual Roles of Monitor during Inflation</td>
<td>382</td>
</tr>
<tr>
<td>18.4.1.4</td>
<td>Redundant Monitor Locking/Unlocking Pair</td>
<td>382</td>
</tr>
<tr>
<td>18.4.1.5</td>
<td>Implementation with Merged Monitor and Control</td>
<td>383</td>
</tr>
<tr>
<td>18.4.2</td>
<td>Fat-Lock Deflation to Thin-Lock</td>
<td>385</td>
</tr>
<tr>
<td>18.4.2.1</td>
<td>Conditions for Lock Deflation</td>
<td>385</td>
</tr>
<tr>
<td>18.4.2.2</td>
<td>Design of Lock Deflation</td>
<td>386</td>
</tr>
<tr>
<td>18.4.2.3</td>
<td>Supports to Lock Deflation</td>
<td>388</td>
</tr>
<tr>
<td>18.5</td>
<td>THREAD-LOCAL LOCK</td>
<td>389</td>
</tr>
<tr>
<td>18.5.1</td>
<td>Lock Reservation</td>
<td>390</td>
</tr>
<tr>
<td>18.5.1.1</td>
<td>Design of Lock Reservation</td>
<td>390</td>
</tr>
<tr>
<td>18.5.1.2</td>
<td>Implementation of Lock Reservation</td>
<td>391</td>
</tr>
<tr>
<td>18.5.1.3</td>
<td>Contention Management on Lock Reservation</td>
<td>392</td>
</tr>
<tr>
<td>18.5.1.4</td>
<td>Discussion on Lock Reservation</td>
<td>394</td>
</tr>
<tr>
<td>18.5.2</td>
<td>Thread-Affined Lock</td>
<td>394</td>
</tr>
<tr>
<td>18.5.2.1</td>
<td>Design of Thread-Affined Lock</td>
<td>395</td>
</tr>
<tr>
<td>18.5.2.2</td>
<td>Inflation Supports to Thread-Affined Lock</td>
<td>398</td>
</tr>
</tbody>
</table>
## 19.1 HARDWARE TRANSACTIONAL MEMORY

### 19.1.1 From Transactional Database to Transactional Memory

### 19.1.2 Intel’s HTM Implementation

## 19.2 MONITOR IMPLEMENTATION WITH HTM

### 19.2.1 Correctness Issues in HTM-Based Monitor

#### 19.2.1.1 Problem without Fallback Handler

#### 19.2.1.2 Problem with Nontransactional Execution

#### 19.2.1.3 Conflict Detection in Transaction

### 19.2.2 Performance Issues in HTM-Based Monitor

#### 19.2.2.1 Introduce Thin-Lock to Transaction

#### 19.2.2.2 Retry Transaction to Alleviate Lemming Effect

## 19.3 CONCURRENT GARBAGE COLLECTION (GC) WITH HTM

### 19.3.1 Opportunities for HTM in GC

#### 19.3.1.1 Object Allocation

#### 19.3.1.2 Root-Set Enumeration

#### 19.3.1.3 Live-Object Marking

#### 19.3.1.4 Dead Object Reclamation

### 19.3.2 Copying Collection

#### 19.3.2.1 To-Space Invariant

#### 19.3.2.2 Current-Copy Invariant with Mutator Transaction

#### 19.3.2.3 Current-Copy Invariant with Collector Transaction

#### 19.3.2.4 Discussion on the Transaction Designs

### 19.3.3 Compacting Collection

#### 19.3.3.1 Idea of Utilizing HTM

#### 19.3.3.2 Find all Heap Slots Pointing to an Object

#### 19.3.3.3 Deal with Potential Data Conflicts

BIBLIOGRAPHY, 425

INDEX, 429
A computing system has traditionally been built on a hardware platform supporting an operating system on which application programs run in the form of machine instructions executed by the hardware. As programming languages evolve, programmers have come to appreciate the benefits that dynamic or managed languages can bring in improving programming productivity. By also offering greater security and software portability, virtual machine has grown to become the preferred environment on which software programs execute nowadays. Today’s state of the art in virtual machine design represents the results of research and development activities undertaken in the past few decades. Those works by and large aimed to improve the implementation of virtual machine with respect to both functionalities and performance. Nowadays, production quality virtual machines are sophisticated and often represent huge implementation efforts accumulated over time. It has become a challenge even for experienced software engineers to understand how a virtual machine performs its work.

I have known Xiao-Feng Li for more than 15 years, since his post at Intel Corporation where he led the development of various compilers and managed runtime systems on Intel platforms. Xiao-Feng was the key contributor to the JVM in the Apache Harmony project. He has also done extensive studies and research work in the design of virtual machines related to Perl, Ruby, JavaScript, and Android. Xiao-Feng’s experience in the engineering and production of VM has allowed him to gain substantial insights into the different areas of VM design, which in turn uniquely positioned him to address the full range of VM-related topics of this book.

Being both a researcher and an engineer, Xiao-Feng has written this book from the unique perspective of a system architect. He emphasizes practical engineering considerations, bringing attention to the interactions among various components, how they work together, and the impact this has on the design of the interface layers. Such details are often not discussed in other books addressing virtual machine. This book also provides detailed figures and code snippets to make the presented ideas easy to understand. This book has become my excellent technical reference on many advanced topics in VM design and implementation. I highly recommend this book to system software developers, especially to those working on managed runtime systems, as it will provide clear answers to many of their questions as they explore the various topics.
By consummating this treatise on VM, Xiao-Feng has made a significant contribution to the design and engineering of virtual machines.

Fred Chow  
Chief Scientist  
Futurewei Technologies, Inc
Preface

This is a book on the design and implementation of virtual machines (VMs) for programming languages such as Java and JavaScript.

Virtual machine, also known as managed runtime system, managed execution environment, and more generally, sandboxing, and the like, has been invented for decades and has been constantly attracting the interests and attention from software researchers and developers due to the important properties that a VM brings to the software, such as safety, productivity, and portability. VMs have become omnipresent in today’s computing systems, from the nodes in IoT (Internet of things), to mobile phones, personal computers, and cloud platforms.

Many of my friends in software-related jobs are curious to learn about the inside of a VM. They frequently ask me questions regarding the VMs they use in their daily work. I found that many of the questions were about common technologies used in a VM, while my friends had difficulties to access the information from existing books and other documents, because those are either mainly focused on the specifications and principles, or are too academic and available in research papers. When my friend Ruijun He, the editor of Taylor & Francis Group, came to me for a book on the topic, I agreed that it would be a good idea to write a book specifically tailored to software developers who have interests in exploring how a VM really “works.”

I have been invited to give lectures on VMs at universities and companies; the lecture notes gradually accumulated into a sequence that appeared as a book. I thought it could be easy to assemble them into a book, but the actual process turned out to be a challenge when I was trying to shape the materials systematically and coherently with both insightful theory support and practical code snippets.

I tried my best to make the book different from the existing literature on similar topics by organizing the contents from the viewpoint of a VM architect who tries to design a VM with a holistic approach. This book tries to organize contents into a consistent framework so that the topics discussed advance step by step, and one algorithm discussed naturally leads to the next. Moreover, this book puts efforts on the parts that are critical to a VM design that are not usually discussed in other documents such as runtime helpers, stack unwinding, and native interface. The algorithms are illustrated in figures and implemented in code snippets, so as to make the abstract concepts tangible and programmable to a system software developer.
The contents of this book were largely finished by the end of 2014. I have been witnessing since then the new VM developments in the industry. However, I did not try to cover various VM implementations, but focused more on the most important technologies that are common to different VMs. I am more than willing to enhance or adjust the contents based on the readers’ feedback. Comments on this book are welcome and can be sent to the publisher or to the author at li@xiaofeng.info.

Xiao-Feng Li
About This Book

Along with the increasingly important runtime engines pervasive in our daily-life computing, there is a strong demand from the software community for an extensive presentation on the design and implementation of modern virtual machines, including the Java virtual machine (JVM), JavaScript engine, and Android execution engine. The community expects to see not only formal algorithm descriptions, but also pragmatic code snippets; it also hopes to understand not only research topics, but also engineering solutions. This book tries to meet the demands by providing a unique description that combines high-level design features and low-level implementations, and it combines advanced topics and commercial solutions.

This book takes a holistic approach to the design of VM architecture, with contents organized into a consistent framework, introducing topics and algorithms in an easily understood step by step process. It focuses on the critical aspects of VM design, which are often overlooked in other works, such as runtime helpers, stack unwinding and native interface. The algorithms are fully illustrated in figures and implemented in easy to digest code snippets, making the abstract concepts tangible and programmable for system software developers.
Chapter 4

Design of Execution Engine

Execution engine is the component that performs the actual operations of the application code. Since the ultimate purpose of application is to execute, execution engine is usually considered the core component of a virtual machine (VM), and the rest components are supportive to the execution engine. Sometimes, the design of the execution engine largely dictates the design of a VM. The two basic execution mechanisms are interpretation and compilation.

4.1 INTERPRETER

It is straightforward to design an interpreter. Once the application code is loaded into memory and parsed into semantic data structures, VM can fetch the code sequence one by one and performs defined operations. The pseudocode for a simple interpreter is as follows:

```
interpret(method)
{
    while( code remains in sequence ){
        read the next code from the sequence;
        if (the code needs more data){
            read more data from the sequence;
        }
        perform actions specified by the code;
    }
}
```

This interpreter should work for many languages. The core in this algorithm is the big loop (called dispatching loop) over the code sequence, which fetches, decodes, and executes every code. The real complexity is hidden in the step of “perform actions defined by the code.” For example when the code is to create a new instance of a class, the interpreter calls into garbage collector to allocate a piece of memory, zero the memory content, initialize the object header (e.g., installing a vtable pointer of the class), and then return the object pointer.
When the code is to invoke a virtual method, the interpreter needs to find out the method address, prepare a stack frame, push the arguments, call the method by recursively interpreting it, and return the result. The invocation of a target method may incur the loading and parsing of the method code if it is not in memory or initialized yet. In other words, all the supportive functionalities of the VM are mobilized and busy working around the interpreter.

The interpreter logic will become less straightforward when the execution flow is intercepted by an exception. Exception leads the control flow into the exception handler that may be out of current method. We will discuss exception handling later in Chapter 11.

4.1.1 Super Instruction

Interpretation usually is slow. One reason among others is its big dispatching loop design that involves branches for every interpreted code. Branches can incur branch miss prediction and instruction cache miss, both of which are expensive. The dispatching also involves lots of memory accesses to read and decode every code. It is easy to think of an acceleration technique that combines two or more codes into one in a preprocessing pass. Then the interpreter can fetch and execute more than one code at a time thus reduce the number of dispatches. The combined code is sometimes called super instruction, quick instruction, or virtual instruction.

For example, the code to add a constant to a local variable in Java bytecode usually needs four bytecodes:

```
//var_1 = var_2 + 2;
1:  iload_1 ; push variable 1 on stack
2:  iconst_2 ; push constant 2 on stack
3:  iadd ; add the stack top two items
4:  istore_1 ; pop stack and store to variable 1
```

If this is a common pattern in a method, we can combine them into one quick instruction with an unused bytecode. Then the interpreter only needs to interpret single bytecode that gives same result as the four.

Since there are only limited number of unused bytecodes, super instructions have limited applicability. An idea is to define different super instructions for different workloads by profiling the workloads and finding out the most efficient bytecode combinations.

4.1.2 Selective Inlining

One another acceleration technique is to compile the execution logic of a bytecode into binary machine code ahead of time in a VM implementation. When that bytecode is dispatched, the interpreter directly transfers its control to the machine code maintained by the VM. Furthermore, the machine codes of multiple bytecodes can be concatenated together so as to eliminate their dispatches. This technique is a workaround of dynamic super-instruction generation and sometimes is called “selective inlining.”

Since the binary machine code has to be generated statically for each bytecode as part of the VM implementation, the VM developer has to make sure the generated binary code is
universal enough for all potential execution contexts. Stitching code is still needed sometimes when two pieces of binary codes cannot directly connect. As a result, the quality of the concatenated code is not high. Just-in-time (JIT) compilation can solve this problem.

4.2 JIT COMPILATION

JIT compilation compiles a piece of application code at runtime into binary machine code, then allows the VM to execute the generated code directly rather than interpret the original piece of application code. It is like treating the entire piece of application code as a single super instruction.

The first question to JIT is how to select the piece of application code to compile. It is natural to consider a method as a compilation unit because of its well-defined semantic boundary. That is why almost all the typical JITs are method based.

4.2.1 Method-Based JIT

Since method is a fundamental language construct, the design of method-based JIT fits into the VM architecture very well. The key data structure is vtable. When JIT is used in a VM, the vtable of a class is installed with function pointers to the virtual methods. For example, to call `ovar.foo()`, the function pointer can be found from `ovar` through its vtable. Vtable data structure is shown in Figure 4.1.

During the class initialization when the methods are not yet compiled, the function pointer to a virtual method actually points to a trampoline that invokes the compiler to compile the virtual method. When the virtual method is called for the first time, the compiler is thus invoked. The compiler compiles the virtual method and installs the compiled binary code address (i.e., the function pointer to the compiled method) into the vtable slot, replacing the original pointer to the trampoline, and then transfers the control to the binary code to finish the first-time invocation. Starting from next time, any invocation on the method will directly go to the compiled code through the vtable. The trampoline code

![Vtable data structure](image-url)
can be released if no one needs it, or be kept for later use again, in case the compiled code is released to save the memory consumed by the code cache. Illustration of trampoline code is given in Figure 4.2.

In this way, the virtual method invocation can be very fast in a few machine instructions. For example, to call $ovar$.foo(), the steps can be expressed in following pseudocode.

\[
\begin{align*}
\text{vtable} &= *ovar; \quad // \text{Get vtable pointer from ovar pointer} \\
\text{foo\_funcptr} &= *(\text{vtable} + \text{foo\_offset}); \quad //\text{get pointer to foo()} \\
(\text{foo\_funcptr})() &= \quad //\text{invoke foo()}
\end{align*}
\]

If it is a VM for X86 processor, the instructions to invoke an virtual method of an object are like the following, assuming eax register holds ovar, the first slot of an object (offset 0) is the vtable pointer, method foo’s function pointer is at offset 16 of vtable.

\[
\begin{align*}
\text{movl} &\quad (%eax), \ %eax \quad //\text{eax now has vtable pointer} \\
\text{movl} &\quad 16(%eax), \ %eax \quad //\text{eax now has foo’s func\_ptr} \\
\text{call} &\quad %eax \quad //\text{invoke foo()}
\end{align*}
\]

Before a method call, all the arguments should have been prepared by the caller (the method that makes the call), so we do not need to prepare them here again. When the last call instruction is executed, X86 processor automatically pushes the return address of the call on the stack, which points to the instruction after the call instruction.

When the method is not compiled, the invocation actually goes to the trampoline as shown below, assuming method foo()’s description data structure is at 0x7001234, JIT compiler’s entrance is at address 0x7005678.

\[
\begin{align*}
\text{pushl} &\quad $0x7001234 \quad //\text{address of foo()’s description} \\
\text{call} &\quad $0x7005678 \quad //\text{address of jit\_compile(method)} \\
\text{jmp} &\quad %eax \quad //\text{eax holds the compiled code entry address}
\end{align*}
\]

The trampoline code first pushes the address of method data structure of virtual method foo(). The runtime stack now has an extra item besides the original state of calling foo(), that is, the arguments and return address. The extra item is then consumed by the call to VM’s function jit\_compile() and then the stack returns to the state of calling foo(). To clean
up the argument by the callee (the function that is called), `jit_compile()` has to be defined to use STDCALL calling convention. Function `jit_compile()` has following prototype.

```c
void* STDCALL jit_compile(Method* method)
```

The function attribute STDCALL should be defined as the VM development environment requires. For example, with GCC, it can be defined like the following, and STDCALL may have to be put in the end of the function prototype.

```c
#define STDCALL __attribute__((stdcall))
```

According to X86 calling convention, the return value of the function call is kept in register `eax`. Here, it holds the entry point address of the compiled binary code. Although the address is supposed to be used as a call target, a `jmp` instruction suffices because the return address has been pushed on the stack by the `call` instruction already. Next time when `foo()` is invoked, the `call` instruction will directly go to the binary code, skipping the trampoline, because the vtable slot has been updated by the compiler to point to the binary code.

When multiple threads want to call the same method and trigger the JIT compilation of the method, VM needs to ensure the mutual exclusion of the compilation on same method. Following is a reduced version of `jit_compile()` implementation in Apache Harmony.

```c
void* STDCALL jit_a_method(Method* kmethod)
{
    uint8* funcptr= NULL;

    /* ensure the class owning this method initialized*/
    class_initialize( kmethod->owner_class );

    /* exclusive compilation */
    spin_lock( kmethod );

    /* if compiled already, return */
    if( kmethod->state == JIT_STATUS_Compiled ){
        spin_unlock( kmethod );
        return kmethod->jitted_code;
    }

    /* now this thread owns the compilation */
    kmethod->state = JIT_STATUS_Compiling;

    if( ! kmethod->is_native_method ){
        funcptr = compile( kmethod );
    } else{ /* a wrapper from jitted code to native */
        funcptr = generate_java_to_native_stub( kmethod );
    }

    /* update the vtable slot with the new funcptr,
```
replacing the original pointer to trampoline */
method_update_vtable( kmethod, funcptr );

/* the method is compiled */
kmethod->state = JIT_STATUS_Compiled;
spin_unlock( kmethod );

    return funcptr;
}

The compile() function in the code above fulfills the actual compilation that translates
the application code into machine code.

Note in the trampoline code above, we have largely simplified the code sequence to be
a direct call into jit_method(). In reality, compiling a method may throw exception, or
enter Java code execution and trigger garbage collection (GC), so the procedure from Java
code execution to JIT compiler (written in native code) needs full Java-to-native transition.
Bookkeeping is needed to make sure all the information be well prepared before entering
the native code and be cleaned up after returning from the native code. We leave this dis-
cussion to Chapter 7.

4.2.2 Trace-Based JIT
In recent years, trace-based JIT has attracted lots of attentions. Trace is a snippet of code
path executed at runtime. Trace-based JIT only compiles the code in the specific path and
leaves alone any other code paths that branch off the specific path.

The main motivation of using trace as the compilation unit is to avoid compiling the
cold code so as to reduce the compilation overhead, in both time and space. Method-based
JIT compiles the whole method including both hot and cold code, even if some code may
never be executed. Trace-based JIT profiles the code execution at runtime and only com-
piles the hot code path, which is called “trace.”

Trace-based JIT has to conduct following tasks.

1. Identify and form the trace
2. Compile the trace and cache the binary code
3. Manage the trace adaptively

Since it is the hot execution path, a trace has to be identified at runtime through profiling.
A common way of profiling is to instrument a counter at the potential entrance of a trace.
The counter is incremented every time when the code following the entrance is executed.
When the counter reaches a threshold, the executed code is considered hot.

Depending on the design, there are normally three kinds of places to instrument a
counter: a method prolog, a loop header, and a basic block.

Method-based profiling is usually used in method-based JIT, that is, when the method
is hot enough, the VM can choose to compile it (if it was only interpreted) or to recompile
it with more advanced optimizations (if it has been compiled). Method-based profiling is straightforward to implement because method entrance is always known to the execution engine. But method-based profiling is not enough to identify all the hot codes. Sometimes, the application spends most of its time in hot loop(s) of a method, while the method itself is invoked only a few times, such as the main() method of a Java application. Even if method-based profiling identifies hot methods, the code in the methods may not all be hot.

Loop usually is considered mostly important for the performance optimization of an application, because a time-consuming application usually spends its execution time in loops. Many advanced compilation optimizations have been developed specifically for loop, such as loop invariant hoisting, parallelization, and vectorization. Therefore, it is natural to try loop-based profiling to identify hot code. A loop construct can be identified at compile-time by analyzing the code control-flow structure, or at runtime by profiling the back edges.

Compile-time loop identification requires the VM to build up the control-flow graph of the application code and then traverse the graph in depth-first order. The edge that points to a node that has already been visited is called back edge, which is the indicator of a potential loop structure. Compile-time loop identification may not be suitable for trace-based JIT if the execution engine does not build control flow graph. Another issue is that compile-time analysis may only be able to find iterative loop but hardly find recursive loop.

Runtime loop identification can be easier. A loop can be identified whenever the control flow goes back to the already-executed code, which is then considered the loop header, where a counter can be instrumented. This approach can only be implemented in an interpreter, because it needs to monitor the execution of every branch operation, which includes normal jump, branch, switch, call, return, and exception-throwing. TraceMonkey of Mozilla Firefox uses this approach.

Dalvik VM in Google Android profiles hot code at basic-block level. It instruments a counter in every maximal basic block. Here, basic block is a compiler term referring to the piece of code that has single entry point and single exit point. Maximal basic block refers to the basic block that cannot be bigger, that is, including more instructions makes it no longer a basic block.

Once a piece of hot code is identified, a trace can be formed by recording the operations in its next time execution (i.e., tracing execution) from the entrance, which is the start point of the trace. This process sometimes is called “tracing.” For loop-based tracing, the trace end point is where the control goes back to the start point. For basic-block-based tracing, the end point is the exit point of the basic block. In both approaches, the length of a trace is limited to avoid the execution strays away from the expected path. Tracing process may give up due to some unsupported conditions, such as exception-throwing or entering runtime services.

Loop-based trace may have some intermediate points where the control branches off the hot path. Tracing process only records the actual taken branches at those points during the tracing execution. But in the following rounds of executions, the control may take other branches rather than the ones recorded in the trace. The VM should ensure correct execution in this situation. In other words, the execution should be able to leave the trace at intermediate points.
When recording the trace, the VM also records the conditions that must be met to keep the trace valid. When the trace is compiled, condition-checking code is inserted into the generated code to ensure the conditions be met to follow the trace; otherwise, control flow aborts the trace execution and transfers gracefully to the off-trace path according to the new conditions. The condition-checking code is called “guard” or “side exit.” For example, with the following loop,

```c
for (i = 0; i < n; ++i)
    j += i;
```

The trace pseudocode may look like below,

```c
start_trace (int i, int j):
    ++i;
    temp = j + i;
    guard ( temp not overflow );
    j = temp;
    guard ( i < n );
    goto start_trace (int i, int j);
```

In dynamic typing languages like JavaScript, the variable type can be dynamically changed. The “same” operator such as “+” can have different operations at runtime when the variables’ types change. The trace only records the types in the tracing execution and can become invalid if the types change in later execution. So the trace also needs to guard the specialized types. On the other hand, specialized types enable the trace to apply many compiler optimizations. For example, if the variables in a trace are all small integers, compiler can easily optimize the code with advanced register allocation technique. Otherwise, memory allocation is necessary to accommodate large integers. Actually, one of major motivations of TraceMonkey is based on the observation that the types in most programs do not change frequently, and the specialized types of the trace can cover most of the runtime possibilities.

Side exiting from a trace incurs high overhead. When side exiting becomes frequent, the whole purpose of trace can be compromised. A solution to frequent side exiting is to expand the tracing scope dynamically.

For loop-based tracing, when a guard fails at runtime, the VM checks its position in the trace. If it is at the trace start point, a new trace is recorded. For dynamic-typed language, the new trace is usually same piece of hot code as original trace, but with a new set of specialized types. If the guard fails in the middle of a trace, the VM recognizes a branch in the trace and starts to profile its hotness. When the branch becomes hot enough, a new trace will start from it. A “trace tree” is then formed together with the original trace. The number of traces for branches should be well controlled to avoid “trace explosion.”

For basic-block-based tracing, the traces of basic blocks can be “chained” so as to avoid involving runtime services or the interpreter. That is, when a trace is known to exit to another trace, the control can transfer to the next trace directly. A guard can be inserted to ensure the chaining be valid. Chained traces can also form a trace tree or trace graph.
Loop-based tracing has an advantage that it can inline methods automatically, as long as the methods are in the execution path of the loop trace. Basic-block-based tracing does not usually cross the method boundary, unless the method is extremely simple that can be inlined ad hoc. Neither of them can handle recursive method tracing. Although loop-based tracing can identify the repetitive execution of a recursion, to form the trace for the recursion is challenging. Except tail recursion, a normal recursion has two disjoint phases of repetitive execution: one is the “downward iterations” that keeps pushing new method frames on the stack, and the other is the “upward iterations” that pops the frames off the stack. The two phases do not know each other, so the second phase has to know how to pop the frames and feeds the return value to the caller frame. This is very ad hoc and difficult to get right. Even this situation works out, indirect recursion is still an untouched problem where a method calls itself through calling other methods.

A question to trace-based JIT is how the VM knows a trace is compiled. This question is solved in method-based JIT by using vtable that links to either the jitted code or the trampoline when it is not compiled. Trace-based JIT does not have vtable, because trace does not have well-defined unit as method does. Trace-based JIT needs a way to maintain the traces and their status. A straightforward solution is to use a dynamic table that can insert the information of a newly identified trace. Dalvik VM uses hash table that maps the trace start address to the hash index, which sometimes leads to hash conflict hence inaccurate trace status. For example, Dalvik VM stores the profiling counter in the hash entry that will be reset when a new trace is mapped into the same entry. As a result, a cold trace may override the information of a hot trace, thus counteracts the design purpose of trace-based JIT.

To the best knowledge of the author, there is no method-based tracing in trace-based JIT. It is not impossible but not very useful. If a method has a hot loop while the method itself is invoked only a few times, method-based tracing may have no way to discover the hot loop and then compile it. If the method is hot because it is invoked in a hot loop, only compiling the method alone without other part of the loop body may not help the loop’s performance. Method-based tracing may be useful for a dynamic language where the method behavior is mainly determined by the argument types. But in this case, JIT method-based compilation with type specialization can be a better solution.

As of year 2015, all the best-known VMs have ceased to use trace-based JITs, mostly due to inferior performance or incredible design complexity for superior performance. Compared to method-based JIT, the benefit of saving compilation time is either unsubstantial or not critical in many cases. The performance benefit due to runtime type specialization and data instantiation is not specific to tracing, but can also be achieved with type inference or other JIT analysis. Ultimately, trace is not a right level of semantic unit for compiler to fully perform its potential.

4.2.3 Region-Based JIT
Region-based JIT can be regarded as a hybrid of method-based JIT and trace-based JIT. The compilation unit can be a basic-block or bigger unit, but it does not necessarily depend on tracing. Region-based JIT is like as a method-based JIT in a smaller granularity, while it can also leverage the runtime information for type specialization and data instantiation.
For static typing languages like Java, region-based JIT can be useful in highly memory-constrained platform by avoiding compiling the whole method. It is also useful when the method is too big in size and takes too long time to compile. The method can be partitioned into regions and only select regions are compiled. To some extent, the region-based compilation can be regarded as a combination of “outlining” and method-based compilation. Outlining is a compilation technique. It moves a piece of code out of the original method and wrapped it as a new method. The original code is replaced by a method call to invoke the newly formed method. The new method is compiled as in a method-based JIT.

For dynamic typing languages, region-based JIT can apply type specialization while avoiding trace explosion. It is based on the fact that basic block does not involve control flow. Compilation at the basic-block level does not have to deal with all the branches, which reduces the chance of exponential increase of the potentially compiled paths. Still guards are needed for type specialization and data instantiation.

Facebook’s HipHop virtual machine (HHVM) for PHP language implements region-based JIT. It does not employ profiling or tracing but compiles the basic block first time it meets, with the runtime types available to the compiler for type specialization. HHVM calls the specialized code for a region “a tracelet.” Guards are generated at the entry of the compiled region to ensure the input variables have the expected types at runtime; otherwise, the compiler is triggered again to generate a new piece of type-specialized code for newly encountered input types. It chains the compiled pieces of the same region with different type specializations as a linked list to match the runtime actual input types, and a right match triggers the trace execution. In the end of the list is a trampoline to trigger a new trace compilation when no matched trace is found in the list. HHVM calls the traces of the region “parallel tracelets.” Parallel tracelets virtually extend the guard code to be a sequence of conditional branches to trigger either a matched tracelet execution or a non-matched tracelet compilation.

Dalvik VM’s trace-based JIT can be considered to be a region-based JIT to some extent.

4.3 RELATION BETWEEN INTERPRETER AND JIT COMPILER

Although interpreter is usually slower than a JIT, it is still widely used in various VM implementations. Interpreter has some benefits such as lower memory footprint and faster application startup time. But those are nonessential. Among other reasons, the major one to use interpreter is its simplicity. When a new language or a new feature of an existing language is introduced, it is much faster to implement in an interpreter than in a JIT compiler. With interpreter, the logic of the new language feature is programmed directly by the developer in the VM implementation language such as C. In other words, the developer has only two dependences:

1. Familiarity with the VM implementation language
2. Understanding of the new language feature, including its syntax and semantics

As a contrast, to implement the new language feature with a JIT compiler, the developer has additional dependences:
1. Familiarity with the target machine Application Binary Interface (ABI) specification
2. Skills in runtime technology to map the new language feature to target machine ABI
3. Skills to develop the compiler to generate the expected target machine code

Consequently, interpreter can help the developers to focus on the new language feature, accelerates the development, and enables fast community adoption.

Another important reason for using interpreter is that some language features are very hard or not worth to implement in a compiler, considering the return on investment, such as,

- Function `eval()` to evaluate a program in the form of a string, which involves the reentrance of the VM
- Statement `throw()` to throw an exception, which needs to unwind the runtime stack hence involves reflection of the VM status
- Operator `new()` to create a new object, which requires support from the memory manager, and may trigger a GC

Even in the most complete compilation-based VM, these features are usually implemented on top of runtime services of the VM, which needs control switch between the jitted code and the VM code. VM code and jitted code usually have different execution contexts, such as different stack frame arrangements for their respective convenience. For example, in jitted code, the stack frames are arranged to enable direct method invocation and return, so it uses the hardware native frame-pointer and instruction pointer (also called program counter), that is, bp and ip registers in X86 architecture. In VM code, the program counter is usually stored in a global variable and points to the current bytecode position that is under execution. The VM may also allocate specific memory area to store the method stack frames. Control switch between the jitted code and the VM code may require the saving and restoration of the execution context. Since interpreter does not have jitted code, nor requires the execution context for jitted code, it is an integral part of the VM. It is straightforward to implement those language features based on runtime services in an interpreter.

Although interpreter is not designed for performance, it does not prevent an interpreter from using compilation for better performance. There are usually two orthogonal ways to introduce a JIT compiler to an interpreter. One is to switch the execution engine between interpretation and compilation back and forth, where JIT is applied to the hot code. The other way is to compile the application code into intermediate representation (IR) such as bytecode and then interpret the IR code. The benefit of this approach comes from the well-formatted IR code, which enables the interpreter’s fast dispatching. This approach is commonly used in today’s interpreter-based VMs. Since it does not generate machine code, the syntax and semantics of IR can be defined with flexibility to encode all the language features while still keeping the interpreter’s portability across different hardware architectures.
4.4 AHEAD-OF-TIME COMPILATION

Although compilation helps performance, JIT works only at runtime, which inevitably adds runtime overhead to the application execution. Ahead-of-time (AOT) compilation tries to reduce the runtime overhead as much as possible by compiling the application code before it is executed.

All the traditional compilers conduct AOT compilation at application development time. But for applications in safe languages that normally run in VMs, AOT compilation is seldom carried out at development time, because that may more or less lose the original benefits of safe language programming. The prebuilt binary code, if without extra security measures, can hardly guarantee the safety and has no way to run across multiple instruction set architectures (ISAs) natively with a single copy.

The AOT compilation is usually conducted after the application’s distribution or deployment. For example, OdinMonkey is an AOT compiler for asm.js language developed by Mozilla Firefox, as part of SpiderMonkey internal implementation. OdinMonkey compiles the application in asm.js language when the application is loaded in the browser before the application starts to execute. Since the application is not compiled before it is loaded into the browser, it keeps the same benefits as JavaScript in safety and portability, which is essential for web applications.

Asm.js is a subset of JavaScript so application in it can still be JIT-compiled with IonMonkey, a method-based JIT implementation in SpiderMonkey. The difference is that asm.js has no runtime features such as dynamic typing, exception-throwing, and GC, which virtually makes asm.js no longer a dynamic language, but similar to C language that can be compiled ahead of time. As a matter of fact, asm.js code is usually automatically generated from C/C++ programs. LLVM clang compiles C/C++ code into LLVM bitcode, which in turn can be translated by Emscripten into asm.js code. So asm.js acts more like an intermediate language for the deployment of web applications developed in C/C++.

Google Chrome’s PNaCl (portable native client) technology does not use asm.js as the intermediate language of web applications; instead, it compiles C/C++ web application code into LLVM bitcode and directly distributes the web application in bitcode, which in turn is AOT-compiled when loaded into Chrome.

As a comparison, Google Chrome’s NaCl and Microsoft Windows’ ActiveX technologies compile the web application code into native machine binary code at development time. A natural consequence is that a web application has to be compiled into multiple copies for different ISAs. Since they do not employ safe language for application distribution, these technologies have to provide other security measures such as sandboxing in Chrome for NaCl code, or digital signing the ActiveX code in Windows.

Besides the benefits of portability and safety, there is a deeper reason why AOT compilation is usually not conducted at development time. That is, the dynamic features of safe language may make it very challenging, if not impossible at all, to fully compile an application with AOT compilation. The dynamic features, such as reflection, eval() function, dynamic class loading, dynamic typing, and GC, make some application information only available at runtime while that information is needed for complete AOT compilation.
For instance, safe language usually does not specify the physical layout of an object, which is subject to the discretion of GC at runtime. When AOT compiler compiles the expression related to object field or property access, it does not even know if the object data is consecutive or discrete in memory. There is no way for it to generate native instructions for object data access unless the object layout information is available, or through reflection support that is much slower. JIT compiler has no such problem because it can get all the information from VM and GC at runtime when it generates instructions.

Dynamic class loading also makes AOT difficult. If a class is not loaded during AOT compilation time, there is no way to compile its methods. Dynamic typing is similar. It allows the variable’s type dynamically vary at runtime. If the AOT compiler cannot infer the variable type, there is no easy way to generate efficient code for the variable’s operations.

For these problems, AOT compiler usually generates code to link with some runtime libraries so as to defer them to runtime. An extreme solution is to compile the entire runtime system together with the application code, which virtually bundles the VM into the application package for distribution. This is a typical approach today to distribute HTML5 applications. It does not actually compile the application ahead of time.

To ease AOT compilation, it is common to conduct the compilation in pseudo-runtime state, that is, setting up the runtime state as much as possible while avoiding actual code execution. For example, an AOT compiler may load all the needed classes and gets the object layout information from the target VM. Or the AOT compilation can be conducted after the VM starts and before any code is executed. The VM can shut down when the compilation is finished, if the VM launch purpose is to assist AOT compilation. In pseudo-runtime AOT compilation, the application execution result should not be committed to the system.

Yet another AOT solution is to only compile the code that is possible to be compiled, leaving the not-compiled part to runtime.

Firefox OdinMonkey can do AOT compilation for asm.js code because asm.js virtually removes all the dynamic features of JavaScript. Android application’s intermediate language dexcode keeps certain dynamic features of Java bytecode, Android Runtime (ART) has to conducts AOT compilation on dexcode in pseudo-runtime state. To identify the right classes to compile, ART needs to load the needed classes and hence executes the class initializers with a built-in interpreter during AOT compilation. In other words, the AOT compiler involves almost a full VM.

Since some AOT compilers need to execute the application code, it is interesting to discuss the real boundary between JIT and AOT compilations. They have following differences:

1. AOT compilation is usually conducted without actually executing the application or committing the execution result. In other words, the application is not at “runtime” state. AOT may execute some code of the application, but the reason for the execution is a compromise to make AOT compilation possible, rather than to get the execution result for which the application is developed.

2. AOT compilation does not surely know whether the methods it compiles will or not be executed in an actual run of the application, because it does not have the all
runtime information on the control flow. AOT may have some heuristics or profiling information that can help the method selection. As a comparison, JIT only compiles the methods that are surely to be executed.

3. AOT compilation and application execution are two strictly separated phases. These two phases are not interleaved and can be separated in both time and space. In other words, when needed, the AOT phase can save the compiled result in one place, and later the execution phase can use the result in another place and does not need to compile again. The AOT compilation can be conducted at application development time, deployment time, installation time, launch time, and so on, depending on the design of the VM, the language, and the application.

The major motivation for AOT compilation is to save the runtime overhead incurred by JIT in time and space while still keeping the performance benefit over interpreter. But AOT may not be able to implement all the optimizations available to JIT, because of the nonruntime nature. For instance, type specialization for dynamic language requires the compiler know the runtime types of the variables, which is not usually possible in AOT. Another example is on runtime safety enforcement. Java VM (JVM) requires to ensure the access to an array element to be always within the array bound, so an array bound checking is enforced before any array element access. If the compiler knows that the access is always within the array bound, it may eliminate the redundant bound checking. The element index and array length are usually much easier to obtain at JIT time than at AOT time.

However, AOT compilation can enable some heavy-weighted optimizations that are usually not used in JIT, due to the excessive runtime overhead for the optimizations. Long compilation time in JIT may cause user-perceivable stuttering in the application’s execution, so sometimes it has to balance between compilation time and execution time. AOT may not need this tradeoff; hence, AOT can apply optimizations like interprocedural optimizations and whole-application escape analysis that are usually not fully touched in JIT.

Although all the traditional static compilation can be regarded as AOT compilation, they are not usually called this way. AOT compilation—when it is explicitly stated—is usually considered a special form of JIT as a kind of dynamic compilation, rather than a kind of static compilation.

4.5 COMPILe-TIME VERSUS RUNTIME

Compile-time refers to the time when a compiler is compiling. Runtime refers to the time when an application is running. Traditionally, these two phases are decoupled, while in JIT-based VM they are overlapping, because JIT compiles at runtime. A better definition of the terms should correlate the subject and object of the phases.

Assuming program P written in language L is compiled to machine code C, compile-time refers to the time when program P is compiled from L to C, and runtime refers to the time when program P is executed in the form of C.
In a VM, there are two different runtimes. One is the time when program P is executed, that is, program runtime, or application runtime, or simply runtime. The other is the time when the compiled code C is executed, that is, compiled-code runtime. When VM is launched to run program P, it enters application runtime state, but it does not necessarily run any compiled code C yet. When the application code is compiled from L to C, it is at compile-time. Both code compile-time and code runtime happen during the application runtime. Figure 4.3 below illustrates the relation.

The distinction between compile-time and runtime is important to VM developers, because it tells what are available, what can happen, and at what time. For example, in JVM, when an object ovar has been created, and its method foo() is first time invoked, the JIT will be triggered to compile method foo(). In method foo(), there is an object field access to ovar.data as the code below.

```java
int local = ovar.data;
```

The corresponding bytecode seen by JIT can be the following.

```java
getfield 2  // load field #2 "data" from object
istore_4   // store the value to local variable
```

When JIT generates native machine code, the object is already created, and the address, say 0x00abcd00, can be got by JIT when it compiles the bytecode. But JIT should not generate the code for “getfield 2” like below,

```java
// Assuming "data" field is at object offset 0x10
// from the object start address, i.e., at 0x00abcd10,
// since 0x00abcd10 = 0x00abcd00 + 0x10
movl 0x00abcd10, %eax   //copy "data" content to eax. Wrong!
movl %eax, $16(%esp)   //copy eax value to local stack
```

The code sequence is incorrect to access ovar.data directly at 0x00abcd10. The reasons are the followings.
1. Although object ovar’s address is 0x00abcd00 at the compile-time of the bytecode, its address can be different at runtime of the compiled code, because the object can be moved by garbage collector.

2. Although method foo() is compiled due to its invocation upon object ovar, ovar is only an instance of a class, say kclass, that may have other instances created. Method foo() can be invoked upon those other instances.

Actually, although object ovar is the one that triggers foo() compilation, it may not even be the first object that invokes the compiled code of foo(). In a multithreaded application, another thread may invoke foo() right after the compiled code address is installed in the vtable of kclass, before the thread that triggers the compilation starts to run foo()’s compiled code. So the right code sequence generated should be as follows.

```plaintext
// Assuming ovar is stored at stack offset 0x20
// from stack top (saved in register esp).

movl $0x20(%esp), %eax  //copy "ovar" to eax
movl $0x10(%eax), %eax  //copy "ovar.data" to eax
movl %eax, $16(%esp)    //copy eax value to local stack
```

Another example is to invoke the virtual method of an object ovar, such as,

```plaintext
ovar.foo();
```

The corresponding bytecode sequence can be the following.

```plaintext
aload_0  //load ovar to stack
invokevirtual #16 //invoke ovar.foo()
```

At compile-time, JIT knows the current object ovar’s class kclass’ vtable address (say 0x00001000). At the known offset (say 0x10) of the vtable, JIT can find foo()’s entry point (say 0x00002000). But JIT cannot generate instruction to directly call the entry point like below, even if the compiled code never moves.

```plaintext
call 0x00002000  //invoke kclass’ foo() method
```

The reason is, at runtime, the actual object pointed by ovar may be an instance of a subclass of kclass, say sclass, and sclass may override kclass’ method foo(). That means, the method foo() known to JIT at compile-time may not be the foo() that is actually invoked at runtime. So the right code generated should try to identify the right method from object ovar’s vtable, as the following code shows.

```plaintext
movl $0x20(%esp), %eax  //copy "ovar" to eax
movl (%eax), %eax         //load vtable pointer to eax
```
Some application runtime information can be used at method compile-time. For example, as we already have seen, the offset of a method in vtable is available at compile-time in JVM. JIT does not need to generate instructions to retrieve the offset every time calling the method, as below.

```
pushl $16       //push method index
pushl $0x20(%esp)  //push “ovar” to stack
call get_vtable_offset //foo()’s offset in eax
movl $0x20(%esp), %ebx  //copy “ovar” to ebx
movl (%ebx), %ebx  //load vtable pointer to ebx
addl %ebx, %eax    //eax now holds foo()’s entry
call %eax   //call ovar.foo()
```

Since the offset of a method in vtable is fixed in JVM once the class is loaded throughout the application’s runtime, it can be used by JIT in method compile-time without any problem at method runtime.

Note the information available at compile-time or runtime is different from language to language. In some dynamic languages, the object properties (or fields) can be added or deleted at runtime, so normally it is impossible to identify fixed positions for the properties in compile-time. For instance, in JavaScript, it is common to use a hash table to map the property names to the values. In this situation, the access function to the property has to be called at runtime to retrieve the value.

The boundary between compile-time and runtime is not as clear as the figure shows. The subtlety is that the two stages are usually interleaved. For example, to compile a method (when this method is under compiling), the compiler may have to execute another method (e.g., class initializer) before it can finish this method compilation.

On the other hand, when the compiled code of a method is executed, it may invoke another method, hence trigger the JIT compilation of that method. So it is very common to see that method A’s compilation triggers method B’s execution, which in turns triggers method C’s compilation, and when again triggers method D’s compilation, and so on. Consequently, the runtime stack of the VM can be interleaved by compilation frames and execution frames.

In a pure interpreter-based VM, we can say it has no compile-time, hence no distinction between program runtime and compiled-code runtime. The whole lifetime of the VM is to execute the application code and is at runtime. That is one reason why VM is also called runtime system.