Method Handle Optimizations for the JVM Instruction *invokedynamic*

by

Shijie Xu

Master of Computer Science, Tsinghua University, 2007
Bachelor of Science in Computer Science, Northwest University, 2004

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Philosophiae Doctor (PhD)

In the Graduate Academic Unit of Computer Science

Supervisor: David Bremner, PhD, Faculty of Computer Science
Examing Board: Kenneth Kent, PhD, Faculty of Computer Science
               Suprio Ray, PhD, Faculty of Computer Science
               Brent Petersen, PhD, Electrical and Computer Eng
External Examiner: Clark Verbrugge, PhD, School of Computer Science, McGill University

This dissertation is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

September 2017

©Shijie Xu, 2017
Abstract

To better support dynamically typed languages on the JVM, JSR292 proposes a new instruction `invokedynamic`, which allows a user-defined runtime method linkage. With this instruction, a Method Handle Graph (MHG), provided by users, is linked to a dynamic method call site, and is responsible to transfer a dynamic invocation to real method implementations. Here, an MHG is a combination of Method Handles (MHs), which are executable references to Java methods and fields with optional method type transformations.

An MHG’s disadvantage is its complex graph structure. First, an MHG is built by users (e.g., dynamic JVM language interpreters), and the JVM does not have sufficient knowledge to better optimize it until runtime. Second, the existence of equivalent MHGs at runtime wastes memory and CPU resources. Third, complex graph structures introduce additional runtime overhead, when the JVM repeats interpreting, Just-In-Time (JIT) compiling, executing, or garbage collecting them. This overhead becomes significant when dynamic method invocations are prevalent.
This dissertation describes a collection of techniques to understand MHGs, and optimize their execution for the invokedynamic instruction. These techniques are a) MHG pattern mining, which collects MHGs on the JVM runtime and mines their patterns (i.e., frequent transformation patterns and equivalent graph chains) offline for future optimization opportunities; b) an MHG equivalency model and an online equivalent MHG Deduplication System (MHDeS) for memory and computing efficiency. MHDeS detects the uniqueness of an MHG about to be created and avoids its creation by using the existing equivalent MH for execution at program runtime; and c) an MHG simplification by dynamic bytecode generation, which includes a dynamic graph compiler (i.e., GraphJIT), and a template system. GraphJIT compiles a Frequently Traversed but Stable Directed Acyclic (FTSDA) graph into an equivalent but simpler graph by selectively fusing graph internal nodes on the bytecode level, and uses the generated graph to replace the original one for execution, while the template system provides bytecode sources of the MHG nodes for GraphJIT.
Acknowledgements

My appreciation and thanks go to my supervisor Prof David Bremner from UNB and Daniel Heidinga from IBM J9 team. Both helped review my work, and provided valuable comments and suggestions for improvement. I also appreciate my colleagues at the IBM Centre for Advanced Studies (CAS Atlantic) lab at the University of New Brunswick. They provided all necessary helps for my research work.

I would like to thanks Prof Kenneth B. Kent for providing me an opportunity to study in the Lab, and Stephen MacKay, for proofreading my dissertation and papers.

Special thanks to my parents Xiaohua Xu and Chungui Wang for understanding and being so patient, when I am not in China.

This work is supported by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program. Furthermore, we would also like to thank the New Brunswick Innovation Fund for contributing to this project. Finally, we would like to thank the Centre for Advanced Studies - Atlantic for access to the resources for conducting our research.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
<tr>
<td>Abbreviations</td>
<td>xvi</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation and Methodology</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Contributions and Outline</td>
<td>7</td>
</tr>
<tr>
<td><strong>2 Background</strong></td>
<td>10</td>
</tr>
<tr>
<td>2.1 Statically Typed vs. Dynamically Typed Languages</td>
<td>11</td>
</tr>
<tr>
<td>2.2 Java and the Java Virtual Machine</td>
<td>14</td>
</tr>
<tr>
<td>2.3 Programming Language Translation</td>
<td>17</td>
</tr>
</tbody>
</table>
2.3.1 Compilation and Interpretation .................................. 18
2.3.2 Comparison ............................................................ 20
2.4 Compilation .............................................................. 21
  2.4.1 Static Compilation .................................................. 22
    2.4.1.1 Frontend ......................................................... 22
    2.4.1.2 Optimizer ....................................................... 23
    2.4.1.3 Backend .......................................................... 25
    2.4.1.4 Inline Caching Optimization ......................... 27
  2.4.2 Dynamic Compilation ............................................. 29
    2.4.2.1 Dynamic Compilation Cost ......................... 31
    2.4.2.2 Dynamic Compilation Decisions ..................... 32
  2.4.3 JIT compilers ...................................................... 34
2.5 Ruby Languages and Interpreter ..................................... 38
  2.5.1 Ruby Language and Interpreter ................................ 39
    2.5.1.1 Matz’s Ruby Interpreter and Rubinius ............ 39
    2.5.1.2 JRuby and Truffle .......................................... 40
  2.5.2 Ruby and Concurrency ............................................ 41
2.6 JSR 292 ................................................................. 43
  2.6.1 Dynamically Typed Languages and JVM ..................... 44
  2.6.2 Method Handle Graphs ........................................... 48
  2.6.3 Call Site and Method Handle Graph ...................... 52
  2.6.4 JSR 292 Implementation ....................................... 52
3 Evaluation Methodology 55

3.1 Evaluation Philosophy and Goals .............................................. 55

3.2 Measurements ....................................................................... 56

3.2.1 Execution Time .................................................................. 56

3.2.2 Memory ............................................................................. 57

3.3 Evaluation Environment .......................................................... 59

3.3.1 Hardware ......................................................................... 60

3.3.2 Software Environment ......................................................... 60

3.4 Extended Computer Language Benchmark Game ...................... 61

4 Method Handle Pattern Mining 64

4.1 Motivation ............................................................................. 64

4.2 Method Handle Pattern Mining ................................................. 65

4.2.1 Transformation Pattern Mining ............................................ 68

4.2.2 Instance Pattern Mining ....................................................... 69

4.3 Implementation ..................................................................... 70

4.3.1 Base Component .............................................................. 71

4.3.2 Pattern Mining Component .................................................. 72

4.3.2.1 Transformation Pattern Mining ....................................... 73

4.3.2.2 Instance Pattern Mining .................................................. 76

4.4 Mining Findings ..................................................................... 78

5 Method Handle Graph Deduplication System 85

5.1 Motivation ............................................................................. 86
6 GraphJIT and Dynamic Bytecode Generation for Method Handle Graphs

6.1 Motivation .......................................................... 121

6.2 Goal ................................................................. 125

6.2.1 Comparison ...................................................... 126

6.3 Related Work ....................................................... 128

6.3.1 Object Inlining .................................................. 128

6.3.2 Method Handle Inlining ...................................... 129

6.3.3 Co-Location ...................................................... 130

6.3.4 Inline Caching .................................................. 132

6.4 Overview .......................................................... 132

6.5 GraphJIT .......................................................... 134

6.5.1 Concepts ......................................................... 134

6.5.1.1 Class Graphs and Concrete Class Graphs ......... 135

6.5.1.2 Object Graphs ............................................... 137

6.5.1.3 FTSDA Principle and Example .................... 138

6.5.1.4 GraphJIT Translation .................................. 139

6.5.1.5 GraphJIT Overview ..................................... 140

6.5.2 Bytecode Generation Procedure ....................... 142

6.5.2.1 Class Declaration Instruction Generation ........ 144

6.5.2.2 Field Fusion ............................................... 144

6.5.2.3 Method fusion ............................................ 145

6.5.2.4 Node Fusion Criteria ................................... 150
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.5.3</td>
<td>Design Overview</td>
<td>152</td>
</tr>
<tr>
<td>6.5.3.1</td>
<td>Engine</td>
<td>156</td>
</tr>
<tr>
<td>6.5.3.2</td>
<td>Engine Transformation</td>
<td>159</td>
</tr>
<tr>
<td>6.5.3.3</td>
<td>Engine Optimizations</td>
<td>162</td>
</tr>
<tr>
<td>6.5.3.4</td>
<td>Cache Hierarchy</td>
<td>162</td>
</tr>
<tr>
<td>6.5.3.5</td>
<td>Class Loading and Bytecode Verification</td>
<td>166</td>
</tr>
<tr>
<td>6.5.3.6</td>
<td>Plugin System and Interface</td>
<td>168</td>
</tr>
<tr>
<td>6.5.3.7</td>
<td>Synchronous and Asynchronous Generation</td>
<td>170</td>
</tr>
<tr>
<td>6.5.3.8</td>
<td>Concurrency</td>
<td>172</td>
</tr>
<tr>
<td>6.5.3.9</td>
<td>Graph Node Selection Policies</td>
<td>173</td>
</tr>
<tr>
<td>6.6</td>
<td>Method Handle Template System</td>
<td>174</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Generated Class Layout in Templates</td>
<td>175</td>
</tr>
<tr>
<td>6.6.2</td>
<td>The Role of Method Handle Template</td>
<td>177</td>
</tr>
<tr>
<td>6.6.3</td>
<td>Template Overview</td>
<td>178</td>
</tr>
<tr>
<td>6.6.4</td>
<td>Template Implementation and Management</td>
<td>179</td>
</tr>
<tr>
<td>6.7</td>
<td>Template System, GraphJIT and JSR292 Integration</td>
<td>183</td>
</tr>
<tr>
<td>6.7.1</td>
<td>GraphJIT</td>
<td>184</td>
</tr>
<tr>
<td>6.7.2</td>
<td>JIT Compilation of the Generated Bytecodes</td>
<td>185</td>
</tr>
<tr>
<td>6.8</td>
<td>Object Cache and Class Cache</td>
<td>190</td>
</tr>
<tr>
<td>6.9</td>
<td>Evaluation</td>
<td>191</td>
</tr>
<tr>
<td>6.9.1</td>
<td>Memory Use Impact</td>
<td>192</td>
</tr>
<tr>
<td>6.9.2</td>
<td>Execution Time</td>
<td>194</td>
</tr>
<tr>
<td>6.9.2.1</td>
<td>Micro-indy CLBG</td>
<td>196</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Method Invocation Instructions .................................. 17
2.2 Method Handle Transformations [86] .......................... 49
3.1 Variables for Execution Time .................................... 57
3.2 Hardware Platform Configuration .............................. 60
3.3 Software for Individual Tasks .................................. 60
4.1 Transformation Pattern Statistics .............................. 79
5.1 MHG Index Key ..................................................... 102
5.2 MHDeS Expense ................................................... 119
6.1 Speedup Distribution ............................................ 197
6.2 GraphJIT Expense ............................................... 211
6.3 Speedups vs. Filters for the Three Worst Tests. .............. 212
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Three-phase Compiler Architecture</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>Illustration for Inline Caching</td>
<td>28</td>
</tr>
<tr>
<td>2.3</td>
<td>Dynamic Compilation at SELF-93</td>
<td>36</td>
</tr>
<tr>
<td>2.4</td>
<td>Global Interpreter Lock</td>
<td>42</td>
</tr>
<tr>
<td>2.5</td>
<td>Illustration for <code>invokedynamic</code></td>
<td>45</td>
</tr>
<tr>
<td>2.6</td>
<td>Method Handle Sample</td>
<td>51</td>
</tr>
<tr>
<td>3.1</td>
<td>Measure Framework</td>
<td>62</td>
</tr>
<tr>
<td>4.1</td>
<td>Equivalent MHG Sample</td>
<td>69</td>
</tr>
<tr>
<td>4.2</td>
<td>Method Handle Mining System (MHMS) Overview</td>
<td>71</td>
</tr>
<tr>
<td>4.3</td>
<td>SMR</td>
<td>81</td>
</tr>
<tr>
<td>4.4</td>
<td>Equivalent MethodHandle Set Chain Length Distribution</td>
<td>83</td>
</tr>
<tr>
<td>4.5</td>
<td>Global length-two EquivSet’s Size Summary</td>
<td>84</td>
</tr>
<tr>
<td>5.1</td>
<td>Disadvantage of Equivalent MHGs on JIT Compilation</td>
<td>88</td>
</tr>
<tr>
<td>5.2</td>
<td>MH keys and MHG key</td>
<td>95</td>
</tr>
<tr>
<td>5.3</td>
<td>MHDeS Overview</td>
<td>99</td>
</tr>
<tr>
<td>5.4</td>
<td>MH Pool Overview</td>
<td>100</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>5.5 Execution Time Performance Comparison</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>5.6 MHDeS impacts memory usage and Number of GCs</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>5.7 GC Pause Time</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>5.8 Pool Purge Interval vs. ExecTime</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>5.9 JIT Compilation Impact</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>5.10 MHG Reduction Productive</td>
<td>118</td>
<td></td>
</tr>
<tr>
<td>6.1 Graph for Traversal</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>6.2 Two Transformations</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>6.3 Dynamic MHG Optimization Layer</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>6.4 Sample Class Graphs</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>6.5 <em>Object Graph</em> and <em>Concrete Class Graph</em></td>
<td>137</td>
<td></td>
</tr>
<tr>
<td>6.6 <em>Object Graph</em> Simplification Illustration</td>
<td>139</td>
<td></td>
</tr>
<tr>
<td>6.7 GraphJIT Overview (<em>Rules</em> module provides users a way to</td>
<td>155</td>
<td></td>
</tr>
<tr>
<td>force some nodes inlining.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.8 Engine Overview</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>6.9 Sample Dependence Tree</td>
<td>161</td>
<td></td>
</tr>
<tr>
<td>6.10 Mapping in Plugins</td>
<td>169</td>
<td></td>
</tr>
<tr>
<td>6.11 Synch and Asynch Generation</td>
<td>170</td>
<td></td>
</tr>
<tr>
<td>6.12 Relation between MHs and Templates</td>
<td>177</td>
<td></td>
</tr>
<tr>
<td>6.13 Action Chain in Template System</td>
<td>179</td>
<td></td>
</tr>
<tr>
<td>6.14 Template System and GraphJIT</td>
<td>184</td>
<td></td>
</tr>
<tr>
<td>6.15 MHG, GraphJIT and TR.jit Compiler</td>
<td>185</td>
<td></td>
</tr>
</tbody>
</table>
6.16 Illustration: Set equivalent to be $G_{\text{root}}$ ........................................ 187

6.17 Illustration: Set equivalent to be the MH Referencing the Transformation Method ................................................................. 188

6.18 GC Pause Time Comparison ................................................................. 193

6.19 ExecTime Overview ........................................................................... 195

6.20 The Built Common MHG for $cal$ ........................................................ 198

6.21 ExecTime Comparison ....................................................................... 199

6.22 Percentage of Reduced MHs ............................................................... 201

6.23 GraphJIT vs. TR JIT ........................................................................ 204

6.24 Speedup vs. JITRatio ......................................................................... 206

6.25 GraphJIT vs. TR JIT ........................................................................ 207

6.26 Execution Time Speedup vs. The Number of MHGs ......................... 208

A.1 Produced Figure without IC ................................................................. 244

A.2 Produced Figure with IC .................................................................... 245
Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSR292</td>
<td>Java Specification Request 292 (Supporting Dynamically Typed Languages on the Java™ Platform)</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collection</td>
</tr>
<tr>
<td>MH</td>
<td>Method Handle</td>
</tr>
<tr>
<td>MHG</td>
<td>Method Handle Graph that is made up of MHs</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-In-Time Compiler</td>
</tr>
<tr>
<td>FTSDA</td>
<td>Frequently Traversed but Stable Directed Acyclic</td>
</tr>
<tr>
<td>MHDes</td>
<td>Method Handle Deduplication System</td>
</tr>
<tr>
<td>GraphJIT</td>
<td>A runtime JIT compiler that simplifies graph structure via bytecode generation</td>
</tr>
<tr>
<td>Bytecode Class Node</td>
<td>A bytecode queue representing a Java class.</td>
</tr>
<tr>
<td>Bytecode Method Node</td>
<td>A bytecode queue representing a Java method.</td>
</tr>
<tr>
<td>Transformation type</td>
<td>A name indicates what transformation that an MH has.</td>
</tr>
<tr>
<td>Method Type</td>
<td>Argument types and return type that a method has</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Programming languages are tools that connect software developers and computers. With programming languages, software developers are capable of abstracting real problems and representing these problems by programs. Yet, the computer can only understand machine instructions, which are 0, 1 bit sequences. Thus in order to run them, these programs have to be compiled from higher level languages to lower level languages, or interpreted directly. Methods for programming language classification can vary. Languages differ from others in their paradigm (e.g., functional and declarative), typing discipline (i.e., dynamic typing and static typing), execution model (i.e., interpretation and compilation), etc. Based on the typing discipline, programming languages can be divided into dynamically typed languages and statically typed languages. For statically typed languages, the types of program constructs, such as variables, expressions, functions, and modules, are declared,
and their checking is conducted during the program compilation phase, while dynamically typed languages are the opposite. Today, well-known dynamically typed languages include JavaScript, Ruby, and PHP, while languages like C++, Java and Java bytecodes are statically typed.

The distinction between dynamically typed languages and statically typed languages is not strict. Many modern programming languages have a mix of both characteristics, and these trends becomes more clear along with the rise of virtual machines (VMs), e.g., Java Virtual Machine, Microsoft .Net Common Language Runtime (CLR), and Smalltalk Virtual Machine. A VM is an abstract computing machine, and it is well-known for its automatic memory management, high performance runtime through Just-in-Time (JIT) compilation and optimization, platform independence, and rich ecosystems. Most VMs (e.g., JVM) can only understand and interpret an intermediate language.

Due to the successes and advantages of JVMs (e.g., garbage collection and JIT optimizations), an increasing number of languages, both dynamically typed and statically typed, are implemented on or migrated to JVMs. For example, many popular dynamically typed languages, such as JavaScript, Ruby, Python, and Clojure, have been implemented on the JVM, thanks to the Java Specification Request (JSR) 223 (Scripting for the Java Platform) and JSR 292 (Supporting Dynamically Typed Languages on the Java Platform).

Owing to the gap between dynamically typed languages and bytecode in-
structions, which are the only language that JVM can understand, there are many “pain points” [86] when implementing dynamically typed languages on the JVM. Therefore, JSR 292 was first proposed in 2009, addressing to these “pain points”. This JSR introduces a new JVM bytecode instruction `invokedynamic`, which allows customized method linkage at runtime. According to JSR 292, a dynamic call site is first un-linked, and linked to a method handle after execution of a bootstrap method, where both method handles and the bootstrap method are provided by language implementers. During runtime, the linked method handle then transfers the execution to real method implementations.

A Method Handle (MH) is a typed, directly executable reference to an underlying method, constructor, field, or similar low-level operation, with optional transformations of arguments or return value [14]. These transformations are quite general, and include such patterns as conversion, insertion, deletion, and substitution. In practice, a method that a method handle references might hold references to other method handles. Therefore, multiple method handles, with reference relationships, compose a directed Method Handle Graph (MHG), where a node represents a method handle instance, and an edge represents the field name of the target method handle in the source method handle. An MHG traversal refers to an operation where the JVM visits the whole or partial graph by executing the root MH or recursively iterating all MHs. The root MH’s execution can be started by invoking its `invokeExact` method, which possibly triggers executions of some or all the
MHs in the graph. The MHG iteration can be either depth-first traversal or breadth-first traversal. In this dissertation, an MH and the MHG that is rooted at that method handle are considered interchangeable.

A method handle represents a transformation of method argument types and return type (method type). According to JSR 292, there are more than 15 predefined method handle transformations, e.g., guarding with test transformation, dropping argument transformation, filtering return value transformation, etc. Thus, the permutation space of method handle composition is extremely large when the number of method handles increases, and this provides optimization opportunities for method handle graphs.

1.1 Motivation and Methodology

This dissertation aims to optimize invokesDynamic by looking at the structure of the method handle graph. The main problems with method handle graphs that this dissertation addresses are:

- Method handle graphs are constructed by dynamically typed languages implementers, and the absence of MHG structure knowledge constrains the maximum of JVM optimizations. This is because the existing JVM optimizations, e.g., Just-In-Time (JIT) compilations, are all general-purpose, and they have been designed and intensively tuned for general Java classes before the appearance of JSR 292. To better support MHG execution, a comprehensive understanding of method handle graphs
would provide more optimization opportunities for the JVM.

- There is still much room to optimize method handle graphs for efficient JIT compilation and graph traversal. A JIT compilation in the JVM is a runtime compilation that translates a frequently executed block (e.g., a method or a block of bytecode) into native code.

  First, a method handle represents a simple transformation method, and it is not economical to launch a complete JIT compilation for a single method handle, considering the startup of a JIT compilation procedure.

  Second, the JIT compilation involves a number of tasks, profiling and compiling individual MHG nodes. The accumulated expense on these tasks increases significantly when the graph size increases.

  Third, MHG traversals are prevalent, and the associated costs at runtime are not trivial. Each time a dynamic method is called, a traversal is triggered on the corresponding MHG, if the MHG has not been JITted. The traversal is still necessary in many scenarios (e.g., marking live objects during Garbage Collection), even after the whole graph has been JITted. This is because JIT compiler only translates MHs into native code and inlines these methods together, but it does not eliminate the MH’s dereference costs. Similarly, some internal graph nodes, as well as boxed data referenced by these nodes, might be far away from the root, but frequently used at runtime. In order to access these nodes, the JVM has to repeat visiting a number of intermediate MH
nodes on the path in the graph. These repeated traversals on the whole
or partial graph during runtime slow down execution performance.

Therefore, this dissertation provides a number of solutions to understand
and optimize method handle graphs for potential *invokedynamic* invocation improvement:

- offline method handle pattern mining for frequent MH transformation
  patterns and equivalent instance patterns;

- an online method handle deduplication during method handle graph
  traversal to reduce equivalent method handles, which are found during
  MHG instance mining; and

- dynamic bytecode generation that converts an MHG into an equivalent
  but simpler graph, so that both the effort of MHG interpretation, MHG
  JIT compilation from bytecode to machine code, and MHG Garbage
  Collection (GC) can be reduced. This dynamic code generation is com-
  pleted with the cooperation of the GraphJIT, which is an independent
  runtime graph simplification system, and a template system in the JSR
  292 package. The work of dynamic bytecode generation for MHGs is
driven by one of the MHG transformation pattern mining results: that
a small number of transformation patterns occur frequently.
1.2 Contributions and Outline

This dissertation provides constructive solutions for the efficiency of the JVM method invocation instruction *invokedynamic* from the method handle graph perspective. The topic of the method handles for dynamic JVM languages has received relatively little study in the past five years, owing to the lack of widespread adopters of the instruction in industry. The contributions of this work are

1. identification of various patterns (i.e., transformation patterns and instance patterns) in method handle graphs for future research work on dynamic JVM languages;

2. the idea of detecting and eliminating equivalent method handle graphs that are about to be created at runtime;

3. GraphJIT, a bytecode JIT compiler, which addresses Frequently Traversed but Stable Directed Acyclic (FTSDA) graphs, and takes optimizations that are done by native JIT compilers. GraphJIT selectively fuses graph internal nodes and simplifies the graph. The GraphJIT provides developers a way to tune their FTSDA applications at runtime; and

4. dynamic bytecode generation that compiles a method handle graph into another equivalent, but simpler, one at runtime, and an in-depth evaluation of this solution (e.g., performance analysis and correlation
analysis between performance speedup and various factors). The compi-
lation of dynamic bytecode generation is from bytecode to bytecode
(i.e., bytecode JIT compilation), rather than from bytecodes to native
codes (i.e., native JIT compilation).

The consequence and impact of this work not only helps improve dynamically
typed language implementation on the JVM, but also provides a new insight
to mitigate the gap between typed and untyped programming languages.

The rest of this dissertation is organized as follows:

- Chapter 2 gives necessary background, which covers type systems in
  programming languages, compilation (static compilation and Just-In-
  Time compilation) and interpretation, and fundamental JSR 292.

- Chapter 3 provides evaluation methodology for the proposed method
  handle optimization. It discusses main measurements used in our eval-
  uation, a JRuby benchmark, and an evaluation framework.

- Chapter 4 presents categories of method handle patterns in an MHG,
  and mining method for these patterns.

- Chapter 5 contributes a technique for and implementation of a runtime
  method handle graph deduplication system. This technique detects the
  uniqueness of a method handle about to be created, and avoids creating
  it if it is equivalent to an existing one that was created previously.
• Chapter 6 describes work for dynamic bytecode generation, which simplifies a method handle graph by converting it into another equivalent but simpler one, for the efficiency of JIT compilation and graph traversal. The bytecode generation is completed with the cooperation of GraphJIT, which performs FTSDA graph compilation on the bytecode level, and a template system that provides template bytecode for individual MHs. In this chapter, both GraphJIT and the template system are discussed separately, and then their integration is also presented for JSR 292 in IBM J9 JVM.

• Chapter 7 makes a summary for the whole dissertation.
Chapter 2

Background

This chapter provides some general background for the dissertation. Section 2.1 introduces dynamically typed languages, and Section 2.2 provides some fundamental background for the Java virtual machine and instructions. Next, programming language execution models (i.e., compilation and interpretation) are introduced in Section 2.3 and both static compilation and dynamic compilation are discussed in Section 2.4. Finally, Section 2.5 and Section 2.6 provide information about Ruby language interpreters, and a concrete background for JSR 292.
2.1 Statically Typed vs. Dynamically Typed Languages

There are two kinds of programming languages: statically typed languages and dynamically typed languages. For a statically typed programming language, its constructs, such as variables, data objects, methods, expressions, and structures, have types, which can be declared implicitly or explicitly at compilation time. Different from statically typed languages, dynamically typed languages do not reveal much type information at compilation time. For example, a number 5 is int type; a “data” string is String type; and a method in Listing 2.1 is (int, String)String type\(^1\) indicating that the method accepts two arguments (i.e., an int type and a String type), and returns another String type.

```
1 public String main(int a, String b) {
2    return "\";  
3 }
```

Listing 2.1: Method Type Sample

There is no strict distinction between dynamically typed languages and statically typed languages. Based on the time for type checking, a language is classified to be statically typed, if the type checking is conducted before program execution, while it is dynamically typed if the type checking is at runtime. The type checking at compilation time is feasible for a statically typed

\(^{1}\)The whole thesis uses Java type format.
language, since the type of most variables and methods has to be declared explicitly (or inferable at compilation time), and assignments of objects to these variables must conform to the type consistency. If any inconsistency is found phase, a type error will be reported. However, type checking is not possible for dynamically typed languages before program runtime, owing to the absence of variable and parameter types. For example, the argument \( a \) for a Python method \texttt{fun} in Listing \ref{lst:python-method} is dynamically typed, and its type can not be determined until the variable is assigned at runtime. Here the variable \( a \) can be any type, such as number, string, list, and object.

\begin{verbatim}
def fun(a):
    a.doSomething()
\end{verbatim}

Listing 2.2: Python Method Sample

Languages, such as C++ and Java, are statically typed. Compilers ensure that variables of such languages are assigned data with correct types, that methods are invoked on the correct receivers, that methods are called with arguments of correct types and that the called method is declared. Contrary to C++ and Java, languages, such as Perl, Python, Ruby and JavaScript, are dynamically typed. Perl is a scripting language; Python and Ruby are scripting languages and are also frequently used in web server-side programming; and JavaScript is a scripting language, which can be run in both web browsers as clients and server side as a standalone application.

In these dynamically typed languages, type checking is not available until program runtime (many of these dynamically typed languages are interpreted...
and they do not have a compilation phase). Due to the absence of type checking during the compilation phase, dynamically typed languages are likely to result in more type errors at runtime. For example, if the variable $a$ in Listing 2.2 is associated with an object that does not have method `doSomething()`, the invocation `doSomething()` would throw exception at runtime.

There is no conclusion which one, dynamically typed language or statically typed language, is superior [81, 78]. Normally, statically typed languages have a number of advantages, e.g., early error detection, better documentation in the form of type signatures, more opportunities for optimization, and better developer experiences. First, the statically typed checking detects the potential errors as early as possible, which makes for less runtime failure; and the types deduced during checking provides abundance of program optimization, which makes statically typed programs faster than dynamically typed programs. For example, one well-known optimization is that a virtual method call (late binding) in many statically typed language compilers can be replaced by a direct method call if the receiver type is deduced to have only one candidate (early binding). The existence of these optimizations during program compilation phase improves runtime efficiency by avoiding unnecessary and time-consuming runtime optimization. Second, type checking also benefits code auto-complete and re-factoring in most modern IDEs, which greatly boosts software development productivity.

In contrast, advocates of dynamically typed languages believe that static typing is too rigid, and that the softness of dynamic languages makes them ide-
ally suited for prototyping systems with changing or unknown requirements, or systems that interact with other systems that change unpredictably (data and application integration) [78]. Dynamic typing makes a program more concise and interactive, and yields better polymorphism. Second, another strong requirement for dynamically typed languages is that its program code can be modified at run-time without recompilation and type checking, which is impossible for statically typed programs. This feature is especially useful for unknown scenarios, e.g. software evolution [59], Internet of Things [31], etc. In these areas, programs are deployed in a frequently changeable environment, and the data schema is not known before development and mutable at runtime.

2.2 Java and the Java Virtual Machine

Java Language  The Java programming language is a general-purpose, concurrent, object-oriented language, and it is intended to let application developers “write once, run anywhere,” which means a compiled Java program can run on all platforms without any code modification or recompilation [60]. The Java language is a high-level language so that developers do not need to be aware of what kinds of machines the program will be deployed upon. The Java programming language was first released in the early 1990s by Sun Microsystems (Sun was acquired by Oracle in 2010) and it is currently one of the most widely used programming languages (top 1 and top 2 in 2014
and 2015 according to TIOBE Index [23]). The language is type safe and statically typed, which means type checking is conducted during compilation phase. There are a total of three Java editions: Java 2 Platform Micro Edition (J2ME) for embedded devices, Java 2 Platform Standard Edition (J2SE) for developing and deploying portable Java applications, and Java 2 Enterprise Edition (J2EE) for developing and deploying enterprise applications.

The Java development toolkit (JDK) consists of the Java Runtime Environment (JRE), and a number of Java tools and libraries, e.g., javac and jstack. JRE provides libraries dependencies for a Java program, and its core component is the Java Virtual Machine (JVM). A Java program is converted to an intermediate representation, called bytecode, which can be interpreted by the JVM.

**Java Virtual Machine** The Java Virtual Machine (JVM) is an abstract computing machine, capable of understanding and interpreting bytecode [71]. It is responsible for hardware and operating system independence, and protecting users from malicious programs.

The JVM contributes to the success of the Java programming language, as well as many other JVM hosted programming languages. The JVM offers three fundamental components—automatic memory management, Just-In-Time (JIT) compilation, and concurrency—to improve Java execution performance and development productivity. The automatic memory management is made up of object allocation and Garbage Collection (GC), which
identifies obsolete objects and deletes them from heap memory without developer interference. JIT compilation speeds up program execution sharply by compiling Java bytecode into native machine code at program runtime. The only language that the JVM interpreter can understand is the bytecode specified in [7]. It does not know anything about the Java programming language. Instead, it only accepts a bytecode sequence as input and interprets each bytecode one by one. Any programming language, statically typed or dynamically typed, is compatible with the JVM as long as the source can be compiled to bytecodes. Before interpretation, the class file is first loaded into memory and verified by the verifier. Then the JVM interprets bytecode instructions, and manipulates memory at run-time.

**Java Virtual Machine Bytecode** A bytecode is an intermediate representation of a programming language, and there are various bytecode specifications (e.g., LLVM IR and JVM IR). The bytecode in the JVM is a typed machine code that supports both complex object types and primitive data types, e.g., integer, float, long, and double. The instruction contains an opcode, followed by zero or more operands. The data type (e.g., int, double, or complex class) of the operands is typically embedded in the JVM instruction [7]. For example, the instruction `areturn` implies the returned object is an object reference, while the performed operand for the instruction `ireturn` is `int` type.

Method invocation is a principle mechanism for composing a programming
language and software module. According to the JVM specification, prior to Java 7, there were four method invocation instructions, as shown in Table 2.1. Except for `invokestatic`, the invocation receiver for other method invocations is indispensable, and the format for these instructions are `pc, opcode, owner name, method name, and method descriptor`. For example, `invokevirtual` is used to invoke an instance’s method, while `invokeinterface` is used for an interface method invocation, which is shown below:

11: `invokeinterface MenuItem.getItemId(): I`

27: `invokevirtual MenuInflater.inflate:(ILandroid/...;)V`

<table>
<thead>
<tr>
<th>Inst Name</th>
<th>Scope</th>
<th>Receiver</th>
<th>Dispatch</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>invokevirtual</td>
<td>class</td>
<td>yes</td>
<td>yes</td>
<td>normal invocation</td>
</tr>
<tr>
<td>invokestatic</td>
<td>class</td>
<td>no</td>
<td>n/a</td>
<td>static method</td>
</tr>
<tr>
<td>invokeinterface</td>
<td>interface</td>
<td>yes</td>
<td>yes</td>
<td>interface</td>
</tr>
<tr>
<td>invokespecial</td>
<td>class</td>
<td>yes</td>
<td>no</td>
<td>constructor</td>
</tr>
</tbody>
</table>

Table 2.1: Method Invocation Instructions

### 2.3 Programming Language Translation

The term language translation refers to a conversion from a source programming language to a target language. The translation is necessary since computing hardware does not understand any high-level programming languages that developers use, while it is inefficient to write a program with low-level languages. Therefore, the program written in a high-level language is always
translated to a lower-level target language that the computing hardware or
VM can understand. Here, the high-level language mostly refers to the widely
used languages, e.g., C++, Java, JavaScript, and Ruby, while the low-level
languages, such as assembly language, machine code, and bytecode, are close
to the machine or interpreters. For some specific purposes, e.g., develop-
ment convenience, some modern compilers can also translate one high-level
language into another high-level language. For example, the compiler devel-
oped in the Google Web toolkit [7] project is capable of translating Java code
into JavaScript code.
There are two kinds of translation technologies: compilation and interpreta-
tion.

2.3.1 Compilation and Interpretation

Compilation is a procedure conducted by a language compiler that reads pro-
gram source code as input and generates a target program code. The target
program code can be either native machine code, bytecode, or other inter-
mediate languages that can be understood by an interpreter. Interpretation
is another kind of programming language translation. During interpretation,
a program, called an interpreter, takes each instruction compiled from the
source code, analyzes it and executes it as it goes. After finishing one line,
the interpreter takes the next line and repeats the same procedure again.
There are multiple variations of interpreters:

**Bytecode Interpreters:** These interpreters normally translate source code
into bytecode and then repeatedly fetch, decode, and interpret bytecode instructions. As interpretation is relatively slow, many bytecode interpreters have embedded JIT compilers, which convert hot traces or method blocks to machine code at runtime. Once JIT compilations complete, the interpretation will be turned into execution mode for these JITted targets yielding performance benefits. Examples of modern bytecode interpreters are JVM interpreters, Rubinius and MRI for the Ruby language, and the CPython VM.

**Abstract Syntax Tree (AST) Interpreters:** Different from bytecode interpreters, an AST interpreter does not generate bytecode. Instead, it converts source code into a tree-like representation, AST, and then follows the tree structure directly or generates native code from the tree via JIT. AST interpreters favor both execution and storage optimization before interpretation, because global program structures are preserved in the tree \[45, 68\]. Additionally, AST interpreters help JIT compilation. For example, Oracle’s new experimental Truffle is an AST interpreter \[101, 17\]. With the feedback (e.g., type information) from the runtime, the interpreter replaces some nodes on the tree with more optimized versions. Furthermore, the tree structure and feedback can be utilized by the Graal JIT compiler for better machine code. Based on Truffle, some other AST interpreters are FastR \[68\] for R language, TruffleRuby \[12\] for JRuby, and ZipPy \[98\] for CPython.

**Self-interpreter** A self-interpreter is a programming language interpreter written in a language which can interpret itself. One case of a self-interpreter
is a meta-circular emulator [26], which is written in an implementation of the same language. Some interpreters for Lisp and Prolog programming languages are also examples of self-interpreters.

2.3.2 Comparison

Generally, compiled code is fast and efficient for execution since it is close to the target platform. Also, the optimization conducted during compilation contributes to the speedup of the compiled program, and this optimization can occur when the compiler checks code syntax, generates an efficient intermediate representation, performs intensive optimization with global information and type system [90]. Compared to compilation, interpretation is much slower and less efficient. However, a direct benefit of interpretation is that it makes the interpreted program more independent from computing hardware, which favors interpretation as an attractive solution for programming language design nowadays.

The main disadvantage of the compilation is that the generated code is not portable. For example, a program built by MSVC in the Windows platform can not run in a Unix platform, because instructions and file formats for both OS are different. In contrast to compilation, interpretation is advocated for software engineering because it is convenient for deployment, the development cycle, and migration, although it is also notorious for its lack of efficiency.

Modern programming languages are implemented by mixing of both tech-
nologies gaining the merits of both. From an engineering perspective, both technologies are balanced in modern programming languages to maximize performance and portability. For example, the Java programming language is an interpreted language, yet, it also involves two kinds of compilation, i.e., Java source code to bytecode compilation, and JIT compilation from bytecode to machine code. In this procedure, the existence of bytecode is motivated from program portability considerations while the JIT compilation from bytecode to native machine code is motivated by runtime performance benefits. Meanwhile, programming debuggers are capable of switching between JITted code inside a running program and raw bytecode.

2.4 Compilation

This section reviews a general compiler’s construction and optimization, with a concentration on the dynamic compilation technologies (also known as Just-In-Time Compilation).

According to the time when compilation occurs, compilation can be either static or dynamic. A static compilation is a process when compilation is completed before execution, while dynamic compilation occurs at program run-time.
2.4.1 Static Compilation

For compilation, a compiler generally has to complete a sequence of tasks, e.g., lexical analysis, optimization, and target code generation. Generally, a compiler is made up of three components (two components in some other literature which coalesces Optimizer and Backend into Backend) [2.4 27].

2.4.1.1 Frontend

This component takes source code as input and carries out language analysis, that includes lexical analysis, language syntax analysis, type checking, semantic analysis, and some initial optimization. During the analysis, the source code is first converted into string tokens, and these tokens are validated according to the language rules. Afterwards, an Intermediate Representation (IR), as input for the Optimizer, is generated.

Many tools are available to assist language analysis, e.g., YACC (Yet Another Compiler-Compiler) for C, ANTLR (Another Tool for Language Recognition) for Java, C++, and C#, and JavaCC for Java. All these tools accept language lexical patterns and generate parsers for the target languages.
2.4.1.2 Optimizer

Based on the IR generated from the frontend, the optimizer applies various optimizations and generates another optimized IR. The purpose of these optimizations is to speed up target runtime and reduce memory required for target execution.

Static Single Assignment (SSA) [87, 44, 35], one form of IR, is widely used in compiler design to find data dependencies. In SSA, each variable is only assigned once, and an existing variable would be split into multiple versions. In case a variable depends on a preceding choice in control flow, a $\phi$-function, which chooses a variable version from multiple control flows, will be used at the point where variable definitions converge.

Dataflow analysis refers to a set of techniques that extract data information along program execution paths. It treats a program’s execution as a series of transformations of the program state, which consists of values of all variables and those associated with the program stack. Therefore, a state with more predecessors must have its values defined by combining the values at the predecessors, using a meet (or confluence) operator.

Two instances of dataflow analysis are reaching definitions and live variable analysis. For the former, the definition of a variable $x$ is a statement that assigns (or may assign) a value to $x$, and this definition is a reaching definition for another control point $p$, if there is a path from the definition of $x$ to the $p$, during which $x$ is not “killed” (i.e., redefined) along the path. The latter calculates whether a variable is “live” at some points. During live variable
analysis, a variable is classified to be live at a point, if its value is still used along some paths in the flow graph starting at that point. This information is collected and can be used during register allocation. For example, it is not necessary to store a “dead” variable in a register, and the register with a dead variable’s value should first be used for a new register allocation request.

Along with dataflow analysis and control flow analysis, a number of architecture independent optimizations, such as code elimination, copy propagation, code motion, and loop optimization, are conducted.

- Code elimination is an optimization that removes code, which does not affect the results of program execution. This code can be expressions or blocks that are never reached during runtime (dead code), or expressions that have multiple duplications (redundant code), or expressions that only affect dead variables. Via code elimination, the target program object’s size is shrunk. To detect redundant code, ASTs (or other expression tree structures) can be used. For example, if the tree contains two identical nodes, it may be worthwhile to evaluate the node only once and to use that value twice instead of two evaluations [27].

- Code motion is an optimization that moves code from one place to another place. One well-known code motion is loop-invariant code motion, which moves expressions (or code blocks) out of the loop as there is no difference if expressions are performed inside the loop repeatedly or outside the loop once. A direct benefit of this code motion is that
the moved blocks are only evaluated only once, rather than multiple times when they are inside of loops. Code motion is made possible by the recognition of the essential order of computation of code fragments. Fragments that have no predecessor in the basic block\(^2\) may be moved into the preceding basic blocks of the region or even further, and fragments that have no successor may also be moved into the following ones [27].

- Copy propagation is another optimization that replaces all uses of a variable \(v\) by its expression (or value) during compilation. For an assignment \(u = v\), the variable \(v\) is assigned to another \(u\). In some circumstances, all \(u\) can be replaced by \(v\) directly, thus eliminating both the assignment and the variable \(u\). One advantage of copy propagation is that it often turns the copy statement into dead code [27].

### 2.4.1.3 Backend

The *Backend* is the last step during compilation. In this component, the optimized IR is translated into a target language, which is compatible with a specific hardware machine or an interpreter. This translation procedure mostly involves a number of resource allocations and symbolic data management for debugging, as well as optimizations that are close to the target platform, such as instruction selections, register allocations and assignments.

\(^2\)A basic block normally refers to a sequence of codes with no branches in except to the entry and no branches out except at the exit.
Instruction selection is an operation that maps the IR program into a code sequence that can be executed on the target platform. This mapping is largely determined by factors such as the instruction set the target platform supports, the IR itself, and the desired code quality.

Register allocation is the most frequently discussed optimization in the *Back-end* layer. It addresses a program’s variable placement over a fixed number of registers, such that two simultaneous live variables are not in the same register. An initial attempt is to convert the allocation problem to be finding of a k-coloring for an undirected Register Interference Graph (RIG)\(^3\), which is a NP-complete problem. Based on the RIG, Chaitin et al. try to reduce the problem size during iterations by removing a node \(n\) that has fewer than \(k\) neighbors, and corresponding edges \([41, 40]\). Later, a number of heuristic approaches are used to find optimal or near-optimal solutions with affordable expenses. For example, Briggs et al. provide optimistic coloring and rematerialization \([36]\), both of which aim to reduce the number of procedures to spill variables, and the cost of spilling values. Lal George and Andrew Appel \([58]\) propose Iterated Register Coalescing (IRC) allocation to reduce the coloring complexity by removing some specific variables, coalescing two variables based on their positions, and variable spilling. Other approaches seek coloring based on heuristic knowledge from the control flow \([79, 39]\), or reducing allocation cost by fast linear scanning of variables in the pro-

\(^3\)In an RIG, nodes are variables, and edges connect variables that interfere with one another (e.g., if one variable is live at a point where the other is defined.
gram [84, 94]. A recent solution is trace-based allocation, which splits compilation into traces based on profiling feedback, and then only focus on the efficient register allocation for hot parts of a program [49, 50]. This approach is relatively novel and is specific to VMs that have embedded profiling module for JIT.

2.4.1.4 Inline Caching Optimization

Inline caching is one of the widely used optimization techniques for object-oriented programming languages. It aims to avoid method resolution during invocation by caching previous method resolution results at the call site. In OO programming languages, an interface or a method of a class can have multiple implementations, and the decision as to which implementation should be used for a call site can not be determined at compilation time, because the receiver type of the method invocation might change at run time. In order to retrieve the right implementation, the linker loads the dynamic dispatch tables (e.g., virtual function table in C++) and looks up real method implementations at run time. This accumulated dynamic resolution cost is significant at program runtime and it becomes more severe in modern dynamically typed languages.

Inline caching can reduce this cost by remembering previous resolution results at the call site. To resolve a method invocation, it first calculates the invocation receiver type at the call site, next looks up a real method implementation from a dispatch table that might be dynamically loaded, and
Figure 2.2: Illustration for Inline Caching
then binds the found implementation to the invocation receiver type at the call site directly if this implementation is frequently used. When the invocation is made on the call site again, the bound method implementation is invoked directly if the new receiver type matches a cached type. Otherwise, the failover routine, that repeats the dispatch table lookup, is called again to find the right method implementation.

Depending on the number of implementations cached at the call site, there are three kinds of inline caching: Monomorphic Inline Caching, Polymorphic Inline Caching (PIC), and Megamorphic Inline Caching [25]. In monomorphic inline caching, there is only a single method implementation remembered at the call site, while more than one implementation is remembered in PIC. A PIC sample is shown at the bottom of Figure 2.2, where two method implementations, `Point.display` and `Square.display`, are remembered at the call site. It is reported that execution time can be reduced by 80% and the cache miss ratio reaches as low as 0.5% for some benchmarks for the Self language with PIC [66]. The term "Megamorphic inline caching" is not used frequently as the first two inline cachings. In some places, it refers to a call site, where receiver types vary widely.

2.4.2 Dynamic Compilation

Dynamic compilation, also known as Just-In-Time (JIT) compilation, is a compilation process that occurs at program runtime rather than prior to execution, called Ahead-Of-Time (AOT) compilation [32]. Different from pre-
viously existing static compilation and interpretation, dynamic compilation involves more goals, which don’t exist in static compilation. For example, frequently executed bytecode methods or sequences during interpretation are profiled first and then compiled into machine codes at runtime. Afterwards, these compiled codes substitute for existing bytecode interpretation since they are more efficient for execution.

Dynamic compilation is beneficial to the translation process since it can speed up program execution. It takes full advantage of program runtime information and platform specification, e.g., receiver type statistics at call sites, run-time platform specification, execution path, and program inputs. Therefore, it provides more opportunities for code optimizations, which are not possible for static compilation.

There are two kinds of dynamic compilations according to their inputs: method-based dynamic compilation and trace-based dynamic compilation. For the former, the source for compilation is a block of code (a method, most of the time), that is frequently invoked, while for the latter, the source is a concrete execution path, which consists of a number of contiguous instructions across blocks.

Different from static compilation, there are two phases to do dynamic compilation: a profiling phase and a compilation phase. For method-based JIT compilation, potential methods are monitored and evaluated to determine their hotness in the profiling phase. Correspondingly, the chosen methods with high hotness are translated in the compilation phase.
2.4.2.1 Dynamic Compilation Cost

Dynamic compilation (Just-In-Time compilation) is a compilation that occurs at a program runtime. It involves all the steps that a static compilation has (e.g., source code analyses, optimization, and target code emission). Similarly, these steps consume computing resources, such as memory and CPU at runtime.

Dynamic compilation is not free, and it faces more challenges than a static compilation has. The JIT compiler has to collect runtime information via profiling, for the choice of a good compilation candidate. This is a major challenge for large enterprise-class applications since a large memory overhead is associated with collecting and processing these large volumes of data. In many cases, the system also has to keep the generated native machine code in memory for efficient retrieval, which increases the memory burden at program runtime. The dynamic compilation itself is also a CPU intensive operation. All of these features contribute to a large memory footprint for JIT compilation.

Both runtime overhead of CPU and memory introduced by the JIT compilation have been debated for a long time. Ian Gartle et al., investigate the start-up overhead (throughput) of the interpreter profiling when large J2EE applications were scheduled on the IBM J9VM [57]. More specifically, they observed that start-up is 1.6 times slower than that JVM without interpreter profiling.
2.4.2.2 Dynamic Compilation Decisions

What to compile? It is necessary to choose a subset of code blocks for JIT compilation, so that runtime performance can be maximized. At runtime, some code blocks are frequently executed, and using the generated native code for execution, instead of interpretation, would result in a considerable performance gain. In this case, the cost of the dynamic compilation itself is negligible, when it is compared to this performance gain. Similarly, JIT compilation of code blocks that are seldom executed at runtime is worthless. Thus, it is common to select a small fraction of code blocks for dynamic compilation, especially in an environment where resources are scarce. These selected code blocks, traces or methods, are thought to be hot (the frequently executed code blocks) and will still be hot after they are compiled.

Based on the candidates for compilation, a JIT compilation can be either a method-based JIT or trace-based JIT. For the former, the compilation candidate is a region that has an explicit boundary (e.g., a method or a code block). An early system was Hansen’s work for Fortran IV, which selectively compiled and optimized basic blocks of loop-like ‘segments’ of codes, based on these codes’ dynamic characteristics [64]. Nowadays, JIT compilers in leading VMs, such as SELF [65], HotSpot Server VM, Jikes RVM [30], and TR in IBM J9, are all method-based. For example, SELF-93 compiles each method with a fast non-optimizing compiler and invokes the optimizing compiler on a subset of the frequently executed (or hot) methods. It is also the first JIT compiler that introduces a recompilation decisions and inline caching.
optimizations. For example, if a method is found to be very hot, the SELF system uses a heuristic method to identify the ‘base’ method on the calling stack, and then inlines the traversed call stack into the base method. These design ideas are borrowed by the later HotSpot JVM and IBM J9 JVM.

In contrast to a method-based JIT, one of the assumptions for a trace-based JIT is that most programs spend the majority of time on repeating loops structures, which can be inside of a method or cross multiple methods. As a result, the JIT compiler in a trace-based compilation always collects the bytecode sequence inside of loops, and translates the hot sequence into native machine codes. Some sample trace-based compilers are Dynamo [33], DynamoRIO [38] (Co-developed by HP and MIT Runtime Introspection and Optimization research Group), HotpathVM [55], and traceMonkey (TM) [54] (Mozilla Firefox JavaScript implementation). Dynamo is a dynamic optimization system, and it selects traces using MRET (most recently executed tail), which associates a counter with a certain start of trace points, such as the target addresses of backward taken branches. This counter is updated, and indicates the hotness of the monitored trace. TM is a JavaScript interpreter that records hot loop traces and generates type-specialized native code. Similar to Dynamo, TM detects the start of a trace by looking at the target of backward branches, and the end of a trace as a jump instruction or a side exit.

When to compile? There are two general mechanisms to determine the triggering time for dynamic compilation: counter and sampling. In the
counter mechanism, an integer is associated with a source block (the source block can be part of a source code or any intermediate representation), and its value is increased each time the source block is executed. A block is thought to be hot and the compilation is triggered, if the counter’s value exceeds a given threshold. The counter mechanism is widely used in many of today’s JIT compilers.

The disadvantage of the counter mechanism is its performance overhead owing to the increment of the counter value. The sampling method works much differently from the counter method in that the hotness of a source block is obtained by counting the method at the top of the execution stack. Compared to the counters mechanism, this solution is much lighter weight, but more complicated because it introduces external interruption of a running program.

2.4.3 JIT compilers

Although dynamic compilation is mostly the same as static compilation, its implementation varies in different languages.

SELF The JIT compiler in SELF-93 involves multiple compilation modes, i.e., fast non-optimizing JIT compilation and slow optimizing compilation for time-critical code. At first, the fast non-optimizing compilation is applied to the initial source code without sacrificing performance. Then later, recompilation of the code that is executed frequently is done with a heavy
optimization compiler. By selecting the source method for heavy optimizing compilation, the performance benefit is maximized. In SELF-93, the main optimizations are type feedback and polymorphic inline caching [65, 66]. The compilation procedure in SELF-93 [65] is shown in Figure 2.3. Once a method for compilation in the stack is found, the compiler first compiles this method, and the old version is discarded at the same time. During the compilation, the compiler marks the restart point (i.e., the point where execution can be resumed) and computes the contents of all live registers at that point. If this computation succeeds, the newly generated method replaces the original non-optimized method on the stack, possibly replacing several non-optimized activation records with a single optimized activation record. Then, if the newly optimized method is not at the top of the stack, recompilation continues with the newly optimized methods callee. In this way, the system optimizes an entire call chain from the top recompilation source down to the current execution point.

**Jikes RVM** The Jalapeno dynamic compiler of Jikes VM has two kinds of compilation modes: baseline and optimizing. The baseline compiler translates bytecode into native code by simulating the JVM operand stack without register allocation. The optimizing compiler, on the other hand, translates bytecode into IR, and applies multiple levels of optimization. Jalapeno also introduces subsystems that perform sampling for method hotness calculation, and builds a cost-benefit model to determine the optimization level during
A method overflows its invocation counter and triggers a recompilation. The system inspects the stack to determine which method to recompile. Then it calls the compiler to generate new code. System replaces old stack frames with the newly generated. In the example, the unoptimized is replaced with the optimized one. System continues until all remaining stacks is optimized.

Figure 2.3: Dynamic Compilation at SELF-93 [65]
recompilation of the code [30, 28].

**HotSpot VM**  HotSpot VM organizes JIT compilation into different ordered optimization phases. For a method to be compiled, the VM converts the method’s bytecode content to Static Single Assignment (SSA) representation, and then applies a fixed number of optimizations to the resultant representation. The number of these optimizations varies according to whether it is acting as client compiler or server compiler. Some optimizations are general, e.g., method inlining, dead code elimination, and common sub-expression elimination, while some others are Java specific (e.g., feedback-directed optimizations, fast instanceof/checkcast, and inlining of potentially virtual calls [22]).

**SpiderMonkey in Mozilla**  SpiderMonkey is Mozilla’s JavaScript engine written in C++. It is used in various Mozilla products, including Firefox. SpiderMonkey contains a JavaScript interpreter, a compiler that translates JavaScript source code to bytecode, a garbage collector, and a Just-In-Time compiler (JIT) that converts JS bytecode to native machine code. The JIT compiler in SpiderMonkey has evolved for a long time, from early TraceMonkey to today’s IonMonkey.

*IonMonkey* is a JIT compiler for JavaScript in Firefox. First, it is a traditional JIT compiler. It translates SpiderMonkey’s bytecode into an intermediate representation using SSA, so that a number of common optimizations, such as type specialization, function inlining, and dead code elimination, can
be used. Second, it focuses on the speed of compilation rather than the depth of JS optimization. As JS is dynamically typed and the type information is only available at runtime, the compiler tries to reduce the compilation latency from the perspective of obtaining type information as soon as possible and fewer type checks by type inference \[21\, 9\].

### 2.5 Ruby Languages and Interpreter

In order to obtain the foregoing measurements for the proposed optimizations on the JVM, a benchmark that is capable of triggering `invokedynamic` is necessary. In this dissertation, the JRuby micro-indy benchmark in Computer Language Benchmark Game (CLBG) \[20\] is chosen because

- JRuby was the first and the only well known adopter of `invokedynamic` in Java 7 (both Nashorn and Lambda began since Java 8);

- it is relatively easy to manage the life of JRuby tests. In the micro-indy benchmark, JRuby scripts are all short standalone with a for-loop iteration, the number of which can be fixed or passed as a script parameter. Thus, we can adjust the number of loops so that the proposed optimization can be fully applied at the runtime.

- Rails application is not considered as benchmark in our case. First, rails is only a special JRuby application at server side, which generates dynamic content to serve requests from clients. It is not generalized
for all aspects of a programming languages. Second, it is not easy to manage a rails application as it is alive forever once started, which means that the optimization done in previous benchmarks impacts the memory behavior of later benchmarks.

2.5.1 Ruby Language and Interpreter

The Ruby programming language is a dynamic, open source programming language with a focus on simplicity and productivity [19]. Although Ruby was originally designed to be general purpose, it is popular in the web backend server with the Rails framework. Ruby scripts run at server-side; accept HTTP requests; and serve them with dynamically generated content according to the information in database and request headers. Nowadays, Ruby on Rails has become a key IT infrastructure in many big companies (e.g., Twitter, GitHub, Hulu).

There are multiple implementations of Ruby interpreters. The most popular interpreters are Matz’s Ruby Interpreter (CRuby), Rubinius (Ruby on the Low Level Virtual Machine) and JRuby (Ruby on the Java Virtual Machine). Some other interpreters, like Ruby Enterprise Edition (REE) and IRonRuby, are no longer continued.

2.5.1.1 Matz’s Ruby Interpreter and Rubinius

This Ruby interpreter (also called CRuby) is named for its inventor, Yukihiro Matsumoto. The Matz’s Ruby Interpreter (MRI) is the first Ruby
interpreter, which was a simple AST interpreter. This means the interpreter interprets each node on the AST. Since version 1.9, the MRI was rewritten to generate bytecode, which was also known as Yet Another RubyVM (YARV) [24]. With the introduction of bytecode and inline method caches in YARV [89], MRI’s performance is increased by twofold.

Rubinius is a C++ implementation of Ruby languages. Similar to MRI, it also generates some intermediate bytecode from the AST tree and then interprets these bytecodes. The main difference between Rubinius and MRI is the bytecode instruction set both use. In Rubinius, the instruction set is LLVM IR while it is the built-in instruction set in the YARV.

### 2.5.1.2 JRuby and Truffle

JRuby [11] is a Ruby programming language on the JVM. It is written in the Java programming language and can be executed on the JVM. Before JRuby 1.7, the JRuby interpreter compiled Ruby script to the JVM bytecode directly using a template compiler as it visited each AST node. Similar to Rubinius, JRuby focuses on bytecode generation from Ruby source code, leaving all optimizations to the JVM. In the latest JRuby version, it is attempting to use some high-level IRs that are close to Ruby source code, instead of JVM, and aims for prior optimizations such as constant folding.

JRuby is the key adopter for the new JVM method invocation instruction `invokedynamic`. For JRuby with Java 7, the `invokedynamic` instruction generation is off by default and can be turned on by the option -
JRuby+Truffle is an implementation of Ruby using the Truffle language implementation framework and Graal JIT compiler. It is the AST interpreter and neither bytecode nor IR is generated any more. During interpretation, the Truffle compiler walks through the AST and interprets it directly.

### 2.5.2 Ruby and Concurrency

Many scripting languages (MRI and Python) use a Global Interpreter Lock (GIL) to simplify the internal designs of their interpreters. A GIL is a mutual-exclusion lock held by a programming language interpreter thread to avoid sharing code and internal interpreter data that is not thread-safe with other threads.

The GIL is a very simple synchronization mechanism to support concurrency in an interpreter. For an interpreter with a GIL-based implementation, although a thread can be mapped to a kernel thread (as shown in 2.4), the thread has to acquire the GIL before it can run, and releases the GIL at some pre-defined points. This implementation isolates the code that might result in inconsistencies in a multi-threaded environment, and simplifies the interpreter implementation, especially when multiple native thread-unsafe libraries are involved. The disadvantage of GIL is that it fails to maximize the performance of multiple threads and multiple cores of the hardware.

The removal of GIL is still a hot topic in dynamic language research areas. A traditional and straightforward solution is a fine grained, but complicated,
locking. Another possible solution is *Transactional Memory*, which uses the idea of transactions, rather than locks, to synchronize threads that execute in parallel and share memory. Remigius and Armin [76, 77] compared multiple solutions, i.e., fine grained locking, Transactional Memory (TM), Hardware Transactional Memory (HTM), and Software Transactional Memory (STM) for the GIL replacement, and found that PyPy with STM improves PyPy with GIL by factors in the range of 1.87X up to 5.96X. Rei et al. from IBM eliminated GIL using HTM for Ruby [80]. Based on the collected transaction aborting ratio, they adjusted memory transaction length on a per-bytecode basis to optimize the likelihood of transaction aborts against the relative overhead of the instructions to begin and end the transactions. A third solution is using JVM version interpreters. In this solution, interpreters (e.g., JRuby) only converts scripts that contains concurrency operations to Java bytecodes, and then JVM handles all concurrency tasks.
2.6 JSR 292

Dynamically-typed languages, such as Python, JavaScript, Ruby, and Groovy, are becoming increasingly prevalent, as they provide high programming flexibility, fast prototyping and agile interactive development. A highly efficient runtime is crucial to support such dynamic languages. To avoid developing a runtime for each language from scratch, there is a trend to port such dynamic languages to the Java virtual machine (JVM), which can take full advantage of the JVMs maturity and highly optimized JIT compiler.

However, there are a lot of “pain points” when expressing dynamically typed languages on the JVM [93, 86]. This is because bytecodes are strongly typed and bytecode generation normally requires type information, which is not available for dynamically typed languages. Thus, a lot of extra work and indirections have to be used in both compilation and runtime to infer the variable types to support dynamically typed languages on the JVM. This extra work (i.e., bytecodes and data structures) not only introduces heavy interpretation overhead, but also complicates representation conversion. For example, this extra work defeats the JVM’s attempts at type prediction and target inlining, which in turn blocks subsequent JIT optimization. Furthermore, dynamically typed languages nowadays typically offer capabilities to modify an existing class’s layout (i.e., adding and removing its fields and methods). As a result, it is necessary to defer classes, fields, and methods resolution at runtime, instead of the existing method linkage mechanism.
Also, it is necessary to adapt a method invocation to a target that has a different signature.

To eliminate these “pain points”, in 2009, the JSR 292 Expert Group introduced a new JVM bytecode instruction, called *invokedynamic*, to call a method only by a name and a method type without requiring the real method implementation.

### 2.6.1 Dynamically Typed Languages and JVM

**Invokedynamic components**  A dynamic invocation in JSR 292 consists of three components: an *invokedynamic* instruction, a user-defined bootstrap method, and a target method that is implemented as a method handle.

The definition of *invokedynamic* is

\[
\text{invokedynamic indexbyte1 indexbyte2 0 0}
\]

according to the JVM specification\[71\]. Both *indexbyte1* and *indexbyte2* determine the method name and the signature of this dynamic call site.

In contrast to the four existing JVM method invocation instructions, this instruction allows users to determine how to link the call site to real method implementations, and none of the receiver type information is required for the invocation.

A bootstrap method accepts a method name and a method type, and returns a *CallSite* with a newly created method handle. It is the developer’s (the
one who adopts the \textit{invokedynamic} instruction) responsibility to determine a bootstrap method’s body. Initially, bootstrap methods are registered to a bytecode class. As shown in Figure 2.5, a bootstrap method \texttt{bsm} is invoked once the JVM interpreter sees an \textit{invokedynamic} instruction. Inside of \texttt{bsm}, a call site \texttt{cs} with a newly method handle is returned. Later, all dynamic invocations to \texttt{cs} are transferred to real method implementations until the call site’s target is reset.

Other method invocation instructions, such as \textit{invokevirtual} and \textit{invokeinterface}, can still be used by dynamic JVM language interpreters. It is up to the interpreter itself to determine which instruction should be chosen for dynamic method invocations. For example, interpreters might prefer \textit{invokevirtual} rather than \textit{invokedynamic}, if all types for a dynamic method invocation are deduced as constant.

\textbf{Method Handle} A method handle is a typed, directly executable reference to an underlying method, constructor, field, or similar low-level oper-
ation, with optional transformations of arguments or return values. These transformations are quite general, and include such patterns as conversion, insertion, deletion, and substitution \[13, 16, 14\].

This definition implies three properties of a method handle, which are:

- **typed** A method handle has a method type, indicating what kinds of arguments and return type the target method has.

- **executable** A method handle can be executed directly by invoking either of its methods: `invoke`, `invokeExact`, and `invokeWithArgument`. Although all these three method executions will result in the same bytecode instruction `invokevirtual`, they have different mechanisms regarding the implementation. For `invokeExact`, it requires a complete exact match between the types of all operands on the JVM stack and the method type that the method handle has. If any inconsistency exists, a type mismatch exception will be reported. For the other two methods, the operands type will be transformed automatically if there is any inconsistency.

- **transformation** Method handle transformations are various. They can be applied to any of the arguments and return type of a method invocation. Some transformations are shown in the Listing 2.3.

```java
1 // Direct method handle
2 boolean direct (A a) {
3     MethodHandle mh = lookup.findVirtual(String.class,
```
4 "startsWith", MethodType.methodType
5 (boolean.class, String.class));
6 String a = "Test";
7 assertEquals(a.startsWith(abc), (boolean) mh.invokeExact(a, abc));
8 return (boolean) mh.invokeExact(a, abc);
9
10 }
11
12 // GuardWithTest handle
13 MethodHandle guard;
14 MethodHandle trueTarget;
15 MethodHandle fallback;
16
17 T guardWithTest(A a, B b) {
18     if (guard.invokeExact(a))
19         return (T) trueTarget.invokeExact(a, b);
20     else
21         return (T) fallback.invokeExact(a, b);
22 }
23
24 // FilterReturn Handle
25 MethodHandle target;
26 MethodHandle filter;
27 T adapter(A... a) {
28     V v = target.invokeExact(a...);
29     return filter(v);
Listing 2.3: Method Handle Transformation Sample

In Listing 2.3 three kinds of transformations, i.e., DirectHandle, GuardWithTestHandle, and FilterReturnHandle, are provided. A DirectHandle does not contain any transformation, and it references a target method that does not contain any child MHs. Therefore, a DirectHandle’s execution is the same as the execution of the referenced method. In the sample, mh’s execution is the same as the execution of the String::startWith method directly. The GuardWithTest handle adapts an invocation to the trueTarget by guarding a test. If the test fails, then the invocation is passed to the fallBack method handle. The FilterReturn handle adapts an invocation to the target method handle by filtering its return value.

A method handle is represented by class MethodHandle in java.lang.invoke package since Java 7. Different JSR 292 implementations should provide their own method handle transformation APIs, which are shown in Table 2.2.

2.6.2 Method Handle Graphs

According to Listing 2.3 a method handle with transformation always comes with other method handles. For example, a GuardWithTest handle always holds references to three other kinds of method handles, i.e., guard handle, trueTarget handle, and fallback handle. Correspondingly, these referred
method handles can also have references to other handles in turn. Thus, these recursive method handle references make up a method handle graph. A Method Handle Graph (MHG) is composed of a number of directed connected method handles. In the graph, a node represents a method handle, and a directed edge represents a reference relationship between two method handles. Each edge is labeled by the name of the source method handle’s field referencing the target method handle. For example, `guardWithTestHandle` always has three out-going edges to three different method handles, and these edges are labeled as `guard`, `trueTarget`, and `fallback` respectively. When walking along an MHG, each edge also represents one `invokevirtual` method invocation.

A method handle chain in an MHG is a path from a method handle to the other method handle, and it represents a series of transformations for a

<table>
<thead>
<tr>
<th>Adapters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>convertArguments</td>
<td>pairwise cast, (un)box, pad/truncate</td>
</tr>
<tr>
<td>dropArguments</td>
<td>ignore N consecutive arguments</td>
</tr>
<tr>
<td>insertArguments</td>
<td>insert N arguments at given location</td>
</tr>
<tr>
<td>permuteArguments</td>
<td>reorder (also, drop and/or duplicate)</td>
</tr>
<tr>
<td>collectArguments</td>
<td>collect N trailing (enter varargs)</td>
</tr>
<tr>
<td>spreadArguments</td>
<td>spread N trailing (exit varargs)</td>
</tr>
<tr>
<td>filterArguments</td>
<td>apply filter to arguments</td>
</tr>
<tr>
<td>guardWithTest</td>
<td>wrapper if-else routine</td>
</tr>
<tr>
<td>foldArguments</td>
<td>call target with arguments, which contain pre-processed result</td>
</tr>
<tr>
<td>catchException</td>
<td>catch exception if target throws exception</td>
</tr>
</tbody>
</table>

Table 2.2: Method Handle Transformations [86]
method invocation.

Figure 2.6 shows an example for GuardWithTestHandle, which wraps an if-else routine. In the Figure, GuardWithTestHandle instance 0xFFF5BA48 has three members, 0xBB00 for guard method handle, 0xE098 for trueTarget field, and 0xE0E8 for falseTarget field. The chain (0xFFF5BA48, 0xBB00, 0xE0CD5398) refers to three transformations for the parameters before passing to the terminal method handle 0xE0CD5398.

A whole or partial MHG traversal represents method resolution. In an MHG, a terminal method is always a direct handle, or a virtual handle, representing a real method implementation. The traversal of the method handle graph is a resolution at the invocation call site to find out all real implementations. On invocation, the JVM interpreter walks along the MHG, and then finally reaches the terminal method handles according to method type and other necessary parameters. The path from the root method handle to the terminal method handle is one resolution trace. Additionally, not all method handles in the graph will be visited during each traversal, while some method handles might be visited multiple times if they are shared by graph paths.

It is possible that an MHG contains cycles, which are not forbidden by JSR292. Based on APIs provided by JSR 292, users can build any MHGs. However, the major risk of a cycle in an MHG is that it might result in infinite loops, which make a method resolution failure. This is because a method-based traversal path on an MHG depends on the given arguments at runtime. For example, a GuardWithTestHandle only transfers the invoca-
tion to its trueTarget child, if the execution of its guard child with the given arguments is true.

2.6.3 Call Site and Method Handle Graph

There are three kinds of method call sites according to the JSR 292: a constant call site, a mutable call site, and a volatile call site. The method handle linked at a constant call site is constant once it is initialized. Contrary to the constant call site, the method handle linked at a mutable call site can be modified via resetting, while the target linked at the volatile call site acts like a volatile variable.

All method handles are constant except those that are linked to the mutable call site or volatile call site. The constant means that members in a method handle do not change at program runtime (the modification by the JIT compiler is not considered here). For the method handles that are linked at mutable and volatile call sites, they keep the reference to the call site itself and the modification is achieved by resetting the call site’s target.

2.6.4 JSR 292 Implementation

Different JVM vendors provide their own implementation of JSR 292. For example, John R. Rose from Oracle [86] presents all initial motivation and ideas for JSR 292, and demonstrates a case study on how inline caching works with invokedynamic, as well as potential directions for optimization. This is
the first work that discusses the new instruction. Thalinger and Rose provide a technique to connect method handles with existing JVM optimizations in HotSpot JVM [93]. In their solution, they argue that the overhead associated with call transition between interpretation and compiled method handle is significant. Therefore, they use the server compiler to inline method handles at both dynamic call site and method handle call site before optimization.

J9 is IBM’s independent implementation of the JVM. In J9, the method handle has three representations for compilation [13, 16]:

- interpretation,

- shared thunk that is shared by multiple method handles having the same transformation, method type, and handling specific meta-data, and

- custom thunk that is an independently compiled method handle graph starting at that method handle.

Here a thunk is a structure that holds an entry address to the JITted native code. The generation of method handle thunks is driven by invocation counts and carefully selected to improve the method handle graph traversal. Different from IBM and Oracle’s implementation, Roussel et al. implement JSR 292 in Dalvik, a virtual machine for Android OS. In their work, they simplified the instruction and method handle implementation to adapt to the Dalvik system, which runs in resource constrained devices [88].
Some other works have also been done to illustrate how to use the *invoke-dynamic* instruction. One of the most famous examples is JRuby, a Ruby language implementation on the JVM [5, 4]. Bodden [34] extends Soot, a framework for static analysis and transformation of Java programs, to support *invokedynamic*. Similarly, Ponge et al. showed how the idea of *invokedynamic* is used in Golo, a dynamic programming language for JVM and helps achieve a favorable performance when compared to other dynamic languages [85].
Chapter 3

Evaluation Methodology

This chapter covers a general evaluation methodology for the proposed method handle pattern mining and optimization. We first describe the philosophy and goals of the evaluation. Then we provide the main measurements used in the evaluation. Next, we explain the evaluation environment (both hardware and software) and the motivation for using JRuby benchmarks. Finally, we show a high level benchmark framework that extends the existing Computer Language Benchmark Test (CLBG) framework.

3.1 Evaluation Philosophy and Goals

Because of the mature of the Java ecosystem and to avoid reinventing the wheel, an increasing number of programming languages, both statically typed and dynamically typed, have been migrated to or fully implemented atop of
the JVM. JSR292 (*Supporting dynamically typed languages on the JVM*) is one of the cornerstone JSRs for dynamically typed languages on the JVM. This JSR introduces a new JVM bytecode instruction *invokedynamic*, bootstrap methods and method handle graphs.

The main goal of this dissertation is to improve efficiency of bytecode instruction *invokedynamic* from method handle graph perspectives; the evaluation aims to check how much improvement can be achieved by the introduced solutions in this dissertation. In the dissertation, general measures are execution time and memory consumption.

### 3.2 Measurements

The two main measurements used in this dissertation are execution time and memory.

#### 3.2.1 Execution Time

The execution time of a benchmark test is a measurement to evaluate how fast a program executes. The execution time is also directly impacted by the hardware (e.g., CPU, memory size, and network). To avoid the gap between different hardware, our tests are scheduled to run on the same server.

The elapsed execution time $\text{Exec.Time}$ of a program is used as a key measurement in this dissertation. It is defined as the elapsed in seconds since the tested program begins. The formal execution time $\text{Exec.Time}$ for a single
thread program atop the JVM can be concluded as Equation 3.1

\[
\text{Exec\_Time} = \sum_{i=0}^{n_1} T_{\text{int}_i} + \sum_{i=0}^{n_2} T_{\text{exe}_i} + \sum_{i=0}^{n_3} T_{\text{gc}_i}
\] (3.1)

while \text{Exec\_Time} is Equation 3.2

\[
\text{Exec\_Time} = \max(\text{Exec\_Time}_i), \quad i \in \text{threads}
\] (3.2)

for a program with multiple threads. Table 3.1 shows variables’ definitions, where global GC is a stop-world GC operation

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_{\text{int}_i})</td>
<td>Time expense on interpreting (i^{th}) bytecodes</td>
</tr>
<tr>
<td>(T_{\text{exe}_i})</td>
<td>Time expense on executing (i^{th}) JITted codes</td>
</tr>
<tr>
<td>(T_{\text{gc}_i})</td>
<td>Paused time when doing a global GC</td>
</tr>
</tbody>
</table>

Table 3.1: Variables for Execution Time

### 3.2.2 Memory

Memory is a key storage device in a computer system. It is used to host a program’s instructions and data at runtime to mitigate the time gap between CPU speed and the speed of the external devices (e.g., disk, network). To run a program, the Operating System (OS) loads its instructions and data (whole or partial) into memory, and then allocates an execution time slot
for its execution. During runtime, the amount of memory required might increase or decrease due to dynamic memory allocation and deallocation.

In programming languages, memory is normally divided into two regions: stack memory and heap memory. The stack is a memory region that is managed by compilers and can be highly optimized during code generation. Operations on the stack memory (i.e., push and pop), can be very fast. In contrast to the stack, the heap is a large region memory that is designed for dynamic memory allocations during runtime. For allocated heap memory, objects are alive until they are recycled manually, or implicitly via GC, or at the termination of the program. For example, C++ and C programs can make heap memory by APIs `malloc` and `new`, and return memory back to the OS by APIs `free` and `delete`.

Many modern programming languages have managed memory. The runtime of these languages first requests a large region of memory from the OS, and then manages these memory regions (e.g., object allocation and deallocation). One key feature in the managed memory is Garbage Collection (GC), which detects dead objects and recycles their memory for future use automatically. A classic GC includes three phases: mark, sweep, and compact. In the mark phase, the objects that are still reachable from threads’ root sets are labeled as “alive,” and then those unmarked objects are identified as “dead” and the corresponding memory regions are reclaimed in the sweep phase. Finally, compaction is done to mitigate memory fragmentation and make a large space for future allocation.
Based on the classic GC, there are many GC variation in production JVMs (e.g., Concurrent Mark Sweep GC in HotSpot, Balanced GC in J9 \[3\], and C4 in Azul JVM \[92\] that is declared to be pauseless). The advantages of GC are clear. It resolves dangling pointer issues and prevents memory leaks without the awareness of developers, and thus improves software engineering productivity. However, the main disadvantage of GC is its stop-the-world pause due to object movements and reference updates at runtime.

As GC in a JVM is a key factor for the performance of an application on the JVM, we use three GC related measurements to indicate memory performance:

- **memory usage**: the amount of memory occupied after the last GC. The memory usage can be recognized as the occupied memory when the tested program becomes stable. The memory usage is the size of occupied memory when a server application is idle status.

- **the number of global Garbage Collections**

- **the total GC paused time**

### 3.3 Evaluation Environment

A complete evaluation environment is made up of hardware environment\[3.3.1\] and software environment\[3.3.2\]
3.3.1 Hardware

<table>
<thead>
<tr>
<th>Components</th>
<th>Configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>4 × Intel Xeon E7520 1.8 GHz</td>
</tr>
<tr>
<td>Cores/Threads</td>
<td>16/32</td>
</tr>
<tr>
<td>DRAM</td>
<td>64 GB DDR3 800 MHz</td>
</tr>
</tbody>
</table>

Table 3.2: Hardware Platform Configuration

We conduct our experiment in the Server shown in the Table 3.2.

3.3.2 Software Environment

There are three levels of software in our evaluation: the Java Virtual Machine (JVM), a Benchmark, and JRuby. In addition to this software, we also used a number of language interpreters and compilers (e.g., Python, GNUPlot, Bash shell, and GCC). The main three software for individual work are shown in Table 3.3.

<table>
<thead>
<tr>
<th>Task</th>
<th>JVM</th>
<th>JRuby</th>
</tr>
</thead>
<tbody>
<tr>
<td>MH Pattern Mining</td>
<td>IBM J9.27</td>
<td>Version 1.7.6</td>
</tr>
<tr>
<td>MHG deduplication</td>
<td>IBM J9.28</td>
<td>Version 9.0.3.0</td>
</tr>
<tr>
<td>MHG Bytecode generation</td>
<td>IBM J9.28</td>
<td>Version 9.0.3.0</td>
</tr>
</tbody>
</table>

Table 3.3: Software for Individual Tasks

The benchmark we use is the Computer Language Benchmark Game [6, 20] (CLBG) JRuby micro-indy, which is used to measure different Ruby
language interpreters’ performance. The benchmark has 32 Ruby tests, and each test has 5 to 150 lines Ruby scripts. Most tests have one or two intensive dynamic method invocations via for-loops or recursive calls. During runtime, both verbose GC and JIT logs are collected and analyzed by the benchmark framework.

The JRuby interpreter is the first adopter of the JVM new method invocation instruction `invokedynamic`. When it runs with Java 7, an option `-Xcompile.invokedynamic=true` is required to enable the new instruction emission. At runtime, the JRuby interpreter emits bytecodes from Ruby source, and then executes these bytecodes on the JVM. Both method handles and `invokedynamic` are only related to dynamic method invocations.

### 3.4 Extended Computer Language Benchmark

**Game**

The original micro-indy’s framework is made up of components: configuration, post process, and monitor system. As the existing framework does not satisfy our requirements for memory measurement, we extend it by 1) the removal of the post process step that processes and uploads collected data; 2) the introduction of two runtime components (i.e., sampling monitor system and analysis); and 3) the introduction of a data statistics component.

The framework can be in either sampling or analysis mode. In the sampling mode, the framework performs a data sampling task (i.e., launching
individual benchmark tests and collecting measurements). Each test in the framework is executed twice (with and without introduced optimization) to make results comparable. Once the sampling completes, the framework dumps the collected measurements to an external JSON file. The analysis mode conducts data statistics and visualization. As shown in Figure 3.1, the master Python process spawns a number of monitor processes in sequence, and each monitor process is in charge of a JRuby process (e.g., when to start a JRuby test, when to stop it, and how to respond if a test times out). Similar to monitor processes, the master process also spawns worker processes, which start external Bash shells obtaining memory measurements, such as JIT compilation data and memory footprints.
Both spawned monitors and workers communicate with the original master process via pipes.

The bridge between sampling mode and analysis mode is a JSON data file, which contains all measurements (except JIT compilation). A sample JSON structure is shown in Appendix A.1.
Chapter 4

Method Handle Pattern Mining

This chapter provides our pattern mining solution for method handle graphs (MHGs). In Section 4.1 we present our motivation for MHG pattern mining. Section 4.2 provides method handle patterns, i.e., transformation patterns and instance patterns. Finally, a pattern mining implementation for these patterns is described in Section 4.3 and our findings are presented in Section 4.4.

4.1 Motivation

As discussed in Section 2.6.1, a method handle (MH) is a typed, directly executable reference to an underlying Java method, constructor, field, or similar low-level operation, with optional transformations of arguments or return values. These transformations are quite general, and include patterns such
as conversion, insertion, deletion, and substitution \cite{13, 16, 14}. A Method Handle Graph (MHG) is a directed graph that is composed of multiple interconnected MHs.

The motivations for MHG pattern mining are to

- better understand method handle transformation patterns that dynamic JVM language interpreters have. The JVM only provides basic APIs to create MHGs and an \textit{invokedynamic} instruction. The language implementers use these APIs to combine various MH transformations for the dynamic method implementations. Thus, the MHG pattern mining on the JVM feeds back these patterns to all language implementers for future optimization.

- provide more runtime optimization opportunities for dynamically typed languages on the JVM. The JVM is a platform for dynamically typed languages. The pattern mining result makes the JVM aware of the MHG patterns that a dynamically typed language has, and this in turn helps the JVM to know what kinds of transformation patterns it encounters frequently and drives them to seek more specific optimizations for these frequent patterns.

4.2 Method Handle Pattern Mining

An MHG is represented by its root, from which all MH nodes in the graph are accessible, and each MHG has only one root. Similarly, an MHG’s sub-
graph is made up of all MH nodes that are reachable from the node in the MHG.
Thus, $G_{mh}(V, E)$ defines the MHG that starts at the MH $mh$, where $V$ is a set of MHs and $E$ is a set of directed edges that connect these method handles together.

An MHG creation involves creations of individual MHs in the graph and edges’ setup among these MHs. The order of MH creations is from bottom to top. That is, the leaf MHs are first created via reflective APIs of the `MethodHandles.Lookup`, and then these MHs are used as parameters for creation of the MHs in the top of the graph. Normally, the creation of an MH on the top of the graph involves real memory allocations and a number of method type checkings.

Since a method handle is a reference to a Java method or field, a single method handle inherently has two characteristics: method type $type$ and transformation type $TT$. An MH’s method type is the arguments and return type accepted and returned by the MH. For example, the $type$ of the MH `cat` in Listing 4.1 is $(String, String)String$, indicating that `cat`’s execution requires two $String$ arguments, the first of which is the receiver, and will return a $String$ object. Similarly, the $type$ for `d0` is $(int, String, String, String)String$. MH `d0` transfers an invocation to `cat` by dropping first two arguments (e.g., 123 and “x”). The transformation type $TT$ is the name,

\footnote{MH `cat` refers to a virtual method call, and thus the first argument is the receiver. MH `d0` is a non-virtual method call, and the first argument for its execution is not the receiver.}
Listing 4.1: Method Handle sample

which is either a method handle’s class type or the API that creates the MH. For example, a transformation type of a method handle created by API GuardWithTest is GuardWithTestHandle.

A method handle chain is a path from the root MH to a terminal MH in a graph, and multiple MH chains may share some sub-chains once they are created. Statically, a node in a chain can be shared in multiple chains as fan-in or fan-out nodes. Dynamically, the JVM interpreter walks a chain and interprets each MH node when the chain is executed. According to the interpretation result of an MHG node, the JVM interpreter might jump to a side branch on the chain. Thus, some MH chains might only be partially interpreted at a time, e.g., either trueTarget or falseTarget of a GuardWithTestHandle is executed at one interpretation.

There are two kinds of patterns in an MHG: transformation pattern and instance pattern, which are discussed in Section 4.2.1 and Section 4.2.2 respectively.
4.2.1 Transformation Pattern Mining

A transformation chain is a sequence of transformation types in an MH chain from the root to a terminal MH of an MHG. Take the MGH in Figure 2.6 on Page 51 for example; two transformation chains are:

\[
\text{MutableCallSiteDynamicInvoker} \rightarrow \text{GuardWithTestHandle} \rightarrow \text{MCSDynamicInvokeHandle} \rightarrow \text{ConstantIntHandle},
\]

and

\[
\text{MutableCallSiteDynamicInvoker} \rightarrow \text{GuardWithTestHandle} \rightarrow \text{MoverHandle} \rightarrow \text{Insert1Handle} \rightarrow \text{DirectHandle}.
\]

The transformation pattern refers to the combination of MHs’ transformation types, and its pattern mining is to find common transformation sub-chains, which frequently occur when given the sub-chain’s length. For example, a common transformation sub-chain for the above two transformation chains is

\[
\text{MutableCallSiteDynamicInvoker} \rightarrow \text{GuardWithTestHandle}.
\]

The common transformation sub-chain has the following attributes. First, common transformation sub-chains can be from an MHG or across MHGs. In an extreme case, the transformation sub-chain’s length is one, and the common sub-chains’ pattern mining becomes locating frequent MH transformations. Second, longer transformation sub-chains are likely to be less frequently shared among multiple MH chains.

The goal of identifying these common transformation sub-chains is to obtain these transformation patterns that the dynamically-typed language imple-
menters use frequently. The JVM, as a platform, only provides its API, and is not aware of the kinds of transformations that the language implementers use.

4.2.2 Instance Pattern Mining

*Instance pattern* refers to equivalent method handles in given MHGs, and the purpose of its mining is to find equivalent MHGs for future runtime MH optimizations, e.g., method handle deduplication. The equivalency of two method handles indicates that the transformations of both MHs and their children are the same. In other words, the MHG starts from the position that two method handles are equivalent. For example, there are three MHGs in the Figure 4.1 and two method handles, $N \theta$ and $N \theta'$, are equivalent. However, the MHG starting at $N'$ is not equivalent to $N \theta$, because both root MHs’ children, target and fallback, are not consistent with each other.
According to the definition of equivalent method handles and empirical knowledge, I use the following four rules to determine the equivalency of two method handles \( mh \) and \( mh' \).

**Rule 1.** *Both \( mh \) and \( mh' \) have the same transformation and method type.*

**Rule 2.** *The child method handles referenced by the same fieldName in \( mh \) and \( mh' \) are also equivalent.*

**Rule 3.** *The referenced functions are the same if both are leaf method handles.*

**Rule 4.** *If present, corresponding boxed data values are also equivalent.*

### 4.3 Implementation

The Method Handle Mining System (MHMS) is a system that is built on top of the J9 JVM to mine both transformation and instance patterns in MHGs. The system is mainly comprised of two components, *base component* and *pattern mining component*, as shown in Figure 4.2. The base component collects and analyzes MHGs from cores and puts them into the MHG pool. On the other side, the pattern mining component fetches MHGs from the pool and conducts pattern mining tasks, including the aforementioned *transformation pattern mining* and *instance pattern mining*. Besides these two components, an API layer is also provided to facilitate tasks in both components. For example, the *chain builder* module in the API layer provides functions that
convert an MHG to transformation chains and instance chains directly, while the Statistics API formats the pattern mining results and generates a report.

**4.3.1 Base Component**

The base component serves as an MHG data producer in MHDS. It is made up of three parts: core generator, graph analyzer, and extension. The core generator is a collection of Python scripts that are integrated with a CLBG framework, and these scripts configure the JVM for JRuby scripts and trigger the core generation when the JVM is about to exit. The extension part consists of a number of APIs, i.e., MHG dumping to external files and graph visualization.

The analyzer follows the core generator, and it restores MHGs in memory from core files. A core file is a program snapshot, e.g., arguments, JVM heap memory, and registers, when it is generated. The analyzer in the base
component checks the corresponding regions in the core and rebuilds MHGs. It works as follows. First, for an object that resides in a specified region in a core, the analyzer iterates all of its call sites. Second, the analyzer checks the linked object at a call site. If it is a method handle object, the analyzer creates a new method handle as the root method handle in the memory and initializes it with attribute information, which is available in the core. These attributes include field names, transformation name, and method type. Third, once a root MH, $mh_{root}$, is created, the analyzer recursively visits all child MHs of the original MH in the core, and creates corresponding MHs and links to the right fields in the $mh_{root}$. This procedure continues until all leaf method handles are reached. After the analyzer finishes, the newly created MHG is put into the MHG pool, which is a file storage system. The implementation of the analyzer depends on two libraries, IBM Direct Dump Reader (DDR) tool and IBM Diagnostic Tools for Java (DTFJ) [8], which are delivered with IBM J9 JVM directly. DDR provides a set of Java interfaces for reading J9 structure blobs from a core file generated by J9. DTFJ encapsulates low level DDR interfaces to load and analyze the core file structure.

4.3.2 Pattern Mining Component

The pattern mining component can be in either transformation mining mode or instance mining mode, which is configured during initialization. This component periodically fetches MHGs from the pool, and performs pattern
mining tasks.

4.3.2.1 Transformation Pattern Mining

The transformation pattern mining aims to find $N$ top frequent transformation chains, the length of which are defined via configuration. The approach in this component is to convert transformation pattern mining into finding frequent sub-sequences by suffix trees. Its procedure is mainly made up of following three steps.

- Dictionary setup. A dictionary is created at the beginning to map a transformation name to an alphabet. The alphabet here starts from ‘a’ and increases by one once a pair (transformation name, alphabet) is inserted into the dictionary.[2]

A chain set, set, is also created at the beginning to keep a tuple (chain, weight), where the chain is a mapped transformation chain (starting from the root MH and ending at a leaf MH) in the graph, and the weight is the number of times that the chain shows up in the graph.

Once an MHG is fetched from the pool, it is divided into multiple transformation chains, and the mapped chains are inserted into the set. For example, a sample transformation chain

\[ \text{MutableCallSiteDynamicInvoker} \rightarrow \text{GuardWithTestHandle} \rightarrow \text{MCS-DynamicInvokeHandle} \rightarrow \text{ConstantIntHandle}, \]

with directory entries

[2] There are in total less than 20 transformations. Thus, the alphabet from ‘a’ to ‘z’ is sufficient.
shown in Listing 4.2

1. MutableCallSiteDynamicInvoker=>‘a’,
2. GuardWithTestHandle=>‘b’,
3. MCSDynamicInvokeHandle=>‘c’,
4. ConstantIntHandle=>‘d’

Listing 4.2: Dictionary

is mapped to a chain $a \rightarrow b \rightarrow c \rightarrow d$ (short by abcd), and it is the abcd that is inserted into set.

A tuple would not be inserted into set, if its chain has already been in the set. In this case, the weight of the corresponding tuple in set will be increased by the weight in the tuple about to be inserted.

- Suffix tree mining. Mining the frequent transformation chain consists of finding $N$ top frequent sub-strings, once the chain length is given. This step builds a suffix tree for every two unique tuples, ($chain_i$, $weight_i$) and ($chain_j$, $weight_j$), in set. The suffix tree is built from $s$, which is a joined string of $chain_i$ and $chain_j$ by the character “#”. Based on the suffix tree, the deepest internal node, which has leaf nodes from both $chain_i$ and $chain_j$ is the longest common sub-string (longest frequent transformation chain).

Besides this deepest internal chain, other transformation sub-chains, the lengths of which are greater than $N/2$ and have been in both $chain_i$ and $chain_j$, are also chosen. These sub-chains are recorded as
(\text{subTransForm}_k, o_k) \ (o_k \text{ is the occurrence of the } \text{subTransForm}_k \text{ in both } \text{chain}_i \text{ and } \text{chain}_j). \text{ Thus, there are in total } o_k \times \text{weight}_j \times \text{weight}_i \text{ for } \text{subTransform}_k \text{ for two } \text{chain}_i \text{ and } \text{chain}_j.

- Statistics. The last step is to make a summary (e.g., indexing common strings by their lengths) and to decode back these strings according to the dictionary. For a mined common sub-string \text{subStrTransForm}, its occurrence is

$$s(\text{subStrTransForm}_k) = \sum_{i,j} o_k \times \text{weight}_j \times \text{weight}_i.$$

In the statistics step, the transformation pattern mining result is presented in two ways. One is to show the top frequent common transformation chains by sorting all chains by their \( s(\text{subStrTransForm}_k) \). The other is to group all found sub-strings by their lengths and perform sorting in terms of \( s(\text{subStrTransForm}) \) in individual groups.

The performance tuning for the pattern mining is not necessary due to its offline mining. At runtime, there are in total \( \binom{M}{2} \) suffix trees built using Ukkonen’s algorithm [95], and the complexity for individual trees is \( O(N) \), where \( M \) is the number of unique chains and \( N \) is the length of a chain.
4.3.2.2 Instance Pattern Mining

The four rules for equivalent MHGs in Section 4.2.2 are implemented in Algorithm 1.

In order to reduce a recursive comparison cost during equivalency detection, two data structures are used in Algorithm 1. One is an equivalent set `EquivSet`, which holds all equivalent MHs, and the other is a global `EquivMap` that is declared as

```java
Map<TransformName, List<EquivSet>> EquivMap;
```

During detection, a candidate MH `a` is compared to an MH `b` from `EquivSet`, the index of which is the same as that `a`, and the `a` is added to the `EquivSet` only if the comparison returns true. Otherwise, a new `EquivSet` is created and appended to the corresponding list in the `EquivMap` after adding `a`.

After detection, all equivalent MHs are placed in the same `EquivSet`, which is indexed by their transformation name. For example, parts of entries in `EquivMap` for Figure 4.1 are shown in Listing 4.3.

```plaintext
GWT => [{N0, N0'}]
FilterReturn_ => [{N4}]
```

Listing 4.3: EquivaMap content
Algorithm 1 Two Method Handles’s Equivalency Detection

MH is short for method handle
TT is Transformation Name.

\(a\): method handle candidate to be added
\(b\): The 1st method handle in a \(equivSet\) of list \(equivMap.get(a.TT)\).

**procedure** \(DETECTION(a, b)\)

\[\]
if \(a.countChild() \neq b.countChild()\) then
    return false
end if

if \(a.isDirectory() \text{ and } b.isDirectory()\) then \(\triangleright \text{ Rule 3}\)
    return \(a.targetMethod().isEqual(b.targetMethod())\)
end if

if \(a.boxDataCompare(b)\) is false then \(\triangleright \text{ Rule 4}\)
    return false
end if

for all MH Field’s name \(f\) in \(a\) do \(\triangleright \text{ Rule 2}\)
    \(MH \ a' = a.getField(f); \ MH \ b' = b.getField(f)\);
    if precheck\((a', b')\) then
        continue \(\triangleright \) a’ and b’ previously compared
    end if
    \(Ta = a'.getTerminalFuns(); Tb = b'.getTerminalFuns()\)
    if \(!Ta.equal(Tb)\) then
        return false;
    end if
    if \(!DETECTION(a', b')\) then
        return false
    else
        \(b'.getEquivSet().add(a')\)
    end if
end for

\(b.getEquivSet().add(a)\)
return true
end procedure

**procedure** \(PRECHECK(a', b')\)

return \(b'.getEquivSet().contains(a')\)
end procedure
4.4 Mining Findings

Transformation Mining Findings  To indicate a transformation chain’s frequency, the measure *Relative Frequency (RF)* is defined as

\[
RF_i = \frac{n_i}{\sum_j n_j}, \quad (4.1)
\]

where \(n_i\) is the frequency of the \(i^{th}\) transformation chain found in all MH chains of MHGs. Based on this definition, the higher the \(RF\) value is, the more frequently this transformation chain occurs.

**Finding 1.** The frequency of the different transformation patterns varies significantly, and a small number of transformation patterns occur much more frequently than the others.

To support Finding 1, the 11th most frequent transformation sub-chains are shown in Table 4.1. In the table, the sub-chain with the largest RF accounts for 10.67% of all identified chains (412 in total), while the 11 most frequent chains only accounts for 1.3%.

The short transformation chains have larger RF values while the longer chains are the opposite. For example, the lengths of the top three transformation chains with largest RF values are either 2 or 3. Besides, the accumulated \(RF\) of the top 5 length-two transformation chains, which are the 1st, 2nd, 4th, 7th, and 8th transformation chains in Table 3, is 68.28% while that of the remaining 214 chains is only 31.72%.
<table>
<thead>
<tr>
<th>RF</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.67%</td>
<td>GuardWithTestHandle → GuardWithTestHandle</td>
</tr>
<tr>
<td>7.695%</td>
<td>GuardWithTestHandle → BruteArgumentMoverHandle</td>
</tr>
<tr>
<td>5.957%</td>
<td>GuardWithTestHandle → GuardWithTestHandle → BruteArgumentMoverHandle</td>
</tr>
<tr>
<td>3.754%</td>
<td>BruteArgumentMoverHandle → Insert1Handle</td>
</tr>
<tr>
<td>3.274%</td>
<td>GuardWithTestHandle → BruteArgumentMoverHandle → Insert1Handle</td>
</tr>
<tr>
<td>2.853%</td>
<td>GuardWithTestHandle → GuardWithTestHandle → BruteArgumentMoverHandle → Insert1Handle</td>
</tr>
<tr>
<td>2.462%</td>
<td>BruteArgumentMoverHandle → DirectHandle</td>
</tr>
<tr>
<td>1.988%</td>
<td>BruteArgumentMoverHandle → PermuteHandle</td>
</tr>
<tr>
<td>1.949%</td>
<td>GuardWithTestHandle → GuardWithTestHandle → FoldNonvoidHandle</td>
</tr>
<tr>
<td>1.498%</td>
<td>GuardWithTestHandle → BruteArgumentMoverHandle → DirectHandle</td>
</tr>
<tr>
<td>1.30%</td>
<td>BruteArgumentMoverHandle → Insert1Handle → AsTypehandle</td>
</tr>
</tbody>
</table>

Table 4.1: Transformation Pattern Statistics
The finding of these transformation patterns would benefit to both language implementers and graph optimizations. Based on the transformation pattern statistics, JVM designers can focus optimizations on these frequently seen patterns. Similarly, these transformation patterns provides us an opportunity to tune our systems (i.e., MHDeS and GraphJIT). By configuring some frequent transformations as filters, we can possibly analyze the correlation between these patterns and system performance speedup.

**Instance Mining Result**  Another two measures, SMR and IMHE, are used to explain instance pattern mining results. The SMR (Saved Memory Ratio) is the ratio of the number of MHs, which can theoretically be eliminated, to the total number of the method handles created during program runtime

$$\text{SMR}(j) = \frac{N_j - S_j}{T_j}$$  \hspace{1cm} (4.2)

where $j$ is a benchmark test name, $N_j$ is the number of MHs in all $\text{EquivSets}$, $S_j$ is the number of equivalency sets $\text{EquivSets}$, and $T_j$ is total number of MHs for the benchmark test $j$. The $T_j$ might not be equal to $N_j$ as some kinds of MHs are filtered during equivalency detection. According to the definition, a higher $\text{SMR}$ indicates that more MHs can be eliminated.

The other measure, IMHE (Inverse of Method Handles per Equivalent Set), is a distribution of equivalent MHs per $\text{EquivSet}$, and it is defined as

$$\text{IMHE}(j) = \frac{S_j}{N_j}.$$  \hspace{1cm} (4.3)
This measure indicates the density of an EquivSet, and the smaller the IMHE, the more equivalent MHs there would be in the EquivSet.

**Finding 2.** A large number of equivalencies exist among method handles.

The distributions of both SMR and IMHE for the CLBG benchmark are shown in Figure 4.3. In the figure, SMRs of different tests are relatively stable, ranging from 25% to 32%. This high SMR indicates that there exists a large percentage of equivalent method handles, and a large room to deduplicate these equivalent MHs. In contrast to the SMR, IMHE varies among different tests (ranging from 5% to 23%), and an MH has 7.4 equivalent method handles on average. This indicates that the preference for
transformation construction varies among different tests. For example, the transformation \textit{Insert1Handle} is the most frequently used transformation in test \textit{eval}, while it is \textit{BruteArgumentMoverHandle} in test \textit{pi}.

\textbf{Finding 3.} Similar to Finding 1, short instance chains are likely to have more equivalent method handles, and the distribution of equivalent sets’ sizes is also uneven even when the chains in these sets have the same length. Figure 4.4 shows the size of an \textit{EquivSet} versus the length of chains in the \textit{EquivSet} for individual tests. According to these figures, short chains have more equivalent chains, and bars on the left are crowded and tall, while they become sparse on the right of the figures. Additionally, the equivalent chain sets’ size distribution is uneven even when the chains’ lengths are fixed. Take length-three equivalent sets in the \textit{printf} in Figure 4.4d for example: three \textit{EquivSets’} sizes exceed 100, while the lengths of the remaining sets are less than 45 (the majority of \textit{EquivSets’} sizes are less than 25).

Figure 4.5 demonstrates the size distribution for length-two \textit{equivSets}. In the figure, the \(x\) axis is \textit{EquivSet’s} id, and the \(y\) axis is an \textit{EquivSet’s size}, i.e., the number of MH chains in the set. In the figure, as many as 93% of \textit{EquivSets’} sizes are less than 30, while the remaining 7% are greater than 30. This uneven distribution, together with the conclusion from Figure 4.4, implies that the optimization of the complementary 7% would be the most economical.

All of above findings are valuable for dynamically typed language implementers and JVM optimizations. For the former, they can leverage these
Figure 4.4: Equivalent MethodHandle Set Chain Length Distribution
results by implementing efficient MHG constructions, and conduct user-level MH optimizations, e.g., deduplication and method handle compilation on user level. For the latter, the JVM, as a general platform for all dynamic languages, can focus its effort on specified MHG optimizations, due to uneven distribution of transformation chains, so that the performance benefits and the cost can be balanced.
Chapter 5

Method Handle Graph Deduplication System

This chapter presents a Method Handle Graph Deduplication System (MHDeS) to identify and eliminate equivalent MHGs that are about to be created at program runtime. In this chapter, the motivation for deduplication, our contributions, and related work are first introduced in Section 5.1, Section 5.2, and Section 5.3, respectively. Then, both theoretical background and system design are provided in Section 5.4 and Section 5.5, respectively. Finally, both evaluation and conclusion are made in Section 5.6.
5.1 Motivation

A Method Handle Graph (MHG) consists of a number of method handles, and transfers a method invocation at a dynamic call site to one or more real method invocations via a number of method type transformations.

Resource Consumption  According to Section 4.4, as many as 28.83% of MHGs created by the JRuby interpreter for the CLBG benchmark are equivalent. The existence of these equivalent MHGs not only wastes memory resources, but also burdens program runtime, owing to memory allocations for MHs and a number of type checkings during MH creations.

JIT and Deduplication  An MHG Just-In-Time (JIT) compilation is an MHG translation from bytecode to machine code, when the number of its invocations exceeds a given threshold. This translation in prevalent compilers (e.g., Testarossa (TR) in IBM J9 [61], Jalapeno in the Jikes VM [28, 30] and Client/Server compiler in HotSpot) normally takes two phases. In J9, an MH is translated into a compiled version, called a shared thunk, if it is frequently interpreted (e.g., its invocationCount exceeds a threshold). To reduce the repeated transitions between compiled MHs and other interpreted MHs, these compiled adjacent MH versions are inlined together as a single native method (i.e., custom thunk) [15] with aggressive optimizations. The generated native method later will be inlined at another MH or a dynamic call site directly. A compiled inlined MH version (i.e., a custom thunk) is

86
exclusively owned by an MHG’s root and never be shared with other MHs, while a compiled MH version (i.e., a shared thunk) might be shared with other MHs.

Owing to the lack of MHG sharing, the abundance of equivalent MHGs constrains the exploitation of JIT compilation. As shown in Figure 5.1, three equivalent MHGs, $a$, $b$, and $c$, are not eligible for JIT compilation in Case 1, because all of their $invocationCounts$ are less than the JIT threshold, i.e., 30. Similarly, three other equivalent MHGs, $a'$, $b'$, and $c'$, are JITted separately as each MHG is executed 30 times, which results in three complete JIT compilations for the same bytecodes and three equivalent compiled versions. Thus, a motivation for equivalent MHG deduplication is to avoid redundant JITted MHGs by substituting $a$ for both $b$ and $c$, so that all $invocationCounts$ can be aggregated, as shown in with deduplication part in Figure 5.1. This would also drive earlier JIT compilation of $a$ since $a$’s $invocationCount$ will more easily reach the threshold.

5.2 Contributions

The challenge of MHG deduplication is how to identify a graph. For traditional deduplication systems in data storage areas, data blocks, divided from data objects, are classified to be equivalent if their hash codes (i.e., fingerprints) are the same. Different from these existing systems, where the calculation of a data block’s hash code is fairly straightforward, it is time
consuming for a graph, as it involves a graph traversal, and its time expense is \(O(N + E)\), where \(N\) is the number of graph nodes and \(E\) is the number of edges in the graph. This traversal expense can reach as high as \(O(N^2)\), which is non-trivial when a number of nodes in an MHG increases significantly at program runtime.

This chapter provides an MHG equivalency model and an online MHG Deduplication System (MHDeS) on the JVM. In the chapter, two kinds of keys, MH key and MHG key, are introduced to uniquely identify an MH’s transformation and an MHG’s transformation, respectively. Based on MH keys and MHG keys, an MHG index key represents the MHG’s root that is about to be created, and a model is provided to determine whether the MHG represented by an index key is unique or not. MHDeS, as an equivalency model implementation in J9 (MHDeS is also portable to other JVMs as it does not use any J9 specific APIs), consists of a Method Handle Pool (MH pool), a

\[\text{Figure 5.1: Disadvantage of Equivalent MHGs on JIT Compilation}\]
purger, a detector, and filters. MHDeS organizes all detected unique MHs in the MH pool. In MHDeS, the detector conducts equivalency detection and the purger periodically removes MHs, which are rarely matched during equivalency detection.

The contributions of MHDeS in this chapter are

- Three keys, i.e., MH key, MHG key, and MHG index key, for graph equivalency detection. An MH key identifies the transformation and method type that the MH has; an MHG key consists of MH keys that are ordered by the graph structure; and an MHG index key is a lightweight structure that encapsulates constructor parameters of an MH and represents a method handle about to be created. Creating an index key is much cheaper than creating the corresponding graph, because of fewer memory allocations and type checks. The model determines two method handle graphs’ equivalency by their MHG keys and index keys.

- A formulated graph equivalency model that determines whether an MHG’s root that is about to be created is unique or not. Based on the model, the complexity of finding equivalent sub-MHGs for a given MHG is only cubic in the MHG’s size. Although this model is built for MHGs, it is also applicable for general directed acyclic graphs.

- MHDeS improves the equivalency model by 1) an MHG index key that avoids creating equivalent MHs at program runtime; 2) a transformation index that speeds up transformation chain lookup and reduces the
MHG comparison space; and 3) a fast-path comparison and an MH pool that prevents time-consuming graph traversal by detecting nonequivalent MHGs as early as possible.

- Quantitative analysis of MHDeS performance in terms of execution time, memory usage, MHG reduction effectiveness and time expense for deduplication.

5.3 Related Work

This section discusses related work, which mainly consists of exact graph matching (graph isomorphism) and keys for graph identification.

5.3.1 Graph Matching

Graph matching refers to a class of computational problems of finding an optimal correspondence between pairs of vertices in graphs to minimize (maximize) their node and edge disagreements (affinities). It is a fundamental topic in computer science, as it is a theoretical basis for areas—such as pattern recognition, computer vision, database, and bioinformatics—where correspondence and similarity between multiple graphs are required.

Graph Isomorphism  Graph Isomorphism (also called exact graph matching) is a bijective mapping between two graphs, so that all attributes of nodes and edges are preserved. The maximum isomorphic subgraph problem refers
to finding subgraph $h_1$ of $g_1$ and subgraph $h_2$ of $g_2$, such that $h_1$ and $h_2$ are isomorphic and there is no other pair of isomorphic subgraphs that have more nodes. The problem of graph isomorphism is in NP, but it is not clear whether it is NP-hard [56], while subgraph isomorphism is known to be NP-complete.

Classic methods to resolve subgraph isomorphism problem are to convert it to a tree search [96, 82, 75, 69] procedure, where various heuristic look ahead techniques are applied to reduce the search space. A standard of such algorithms is the one of Ullmann, which introduces a refinement procedure to eliminate successor nodes in the search tree [96] if none of the paired nodes are found for the neighbors of a successor node. Similarly, other traditional solutions for the common subgraph isomorphism are based on maximal clique detection [37] and backtracking. In McGregor’s backtracking method, successors are ordered, and a good successor is the one that leads to subgraphs that have more than the maximal number of edges in common with subgraphs already found [75]. Based on McGregor’s method and Ullmann’s method, Krissinel et al. use a control parameter, minimal size of common subgraph to be found, to avoid search branches that do not lead to a sensible results from the application point of view [69]. A recent work concentrate on the methods that rely on the properties of graphs. For example, Zhu et al. introduce two passes, matching construction and matching refinement for graph matchings [104]. In the construction pass, an initial matching set is built based on anchors in the graph and heuristic information, and these
matchings are refined to a better one with affordable cost in the refinement phase.

In practice, graph isomorphism can be solved in polynomial time when graphs have special structures, or some constrains are added for node comparison. Eppstein decomposes a planar graph into pieces of small tree-width, and then applies dynamic programming within each piece for matching [51]. Luks converts isomorphism of graphs of bounded vertex degrees into the color automorphism problem for groups with small simple sections in polynomial time, and then each group is solved by divide-and-conquer method [72].

MHG deduplication is relevant to exact graph matching. During deduplication, exact MHG matching is necessary to determine the uniqueness of an MH that is in an index key’s ordered child set, when the MH is not found in the MH pool. The main differences between MHG deduplication and graph isomorphism are

- Directed vs. undirected graph matching. MHG are directed graphs, and this direction helps reduce the comparison space.

- Node attributes and an ordered child set. Individual MHs have attributes (i.e., a method type, a transformation, and boxed data), and the number of children, as well as edges, are determined by these attributes. The MHG’s equivalence is much stricter than normal graph isomorphism, and it requires completely equivalence, so that one can replace the other for execution. Although two MHGs have isomorphic
structure, they are not equivalent, if the corresponding children do not match. For example, MHGs $N_0$ and $N'$ in Figure 4.1 on Page 69 are not equivalent, because their root’s children fallback are not equivalent.

- Graph isomorphism normally refers to an exact graph matching, during which graphs have been created. MHG deduplication refers to a procedure that detects an equivalent MHG that is about to be created during the creation time.

Compared to the common undirected graph matching, MHG matching is much simpler, as structure information in a graph can be fully utilized to avoid unnecessary MH comparisons at runtime.

### 5.3.2 Key for Entity Matching and Graph Matching

The idea of using a key to uniquely identify an entity or a graph is straightforward. In the database area, keys are constructed to eliminate duplicate records or to describe record characteristics. For unreliable data sources, Fan et al. provide a method to identify unreliable data records, based on a concept of Matching Dependencies (MDs), which are defined in terms of similarity predicates and a dynamic semantics \[53\]. Then a subset of MDs, called Relative Candidate Keys (RCKs), is selected to determine what attributes to compare and how to compare, when records are across possibly different relations. For data cleaning, Guha et al. define record keys, and introduce Successive Shortest Paths (SSP) algorithm, specifically the top-k
selection problem, to compare records by key’s distances [63].

Pernelle et al. first propose a concept of graph keys for Resource Description Framework (RDF) data [83]. In their method, a key is a combination of objects’ and data properties defined over an Ontology Web Language (OWL) ontology. Similarly, Fan et al. provide a class of keys for graphs, in terms of graph patterns, to specify topological constraints and value bindings needed for identifying graphs [52]. In their solution, these graph keys might be recursively defined, and can be used for graph isomorphism comparison.

The deduplication solution in this dissertation takes advantage of graph key concepts, and applies them for the graph deduplication. The novelty is that an index key is for an MHG, which has not been created yet, and the creations of an index key is much cheaper than the corresponding graph creation.

5.4 Theoretical Model

Equivalent MHGs are graphs that have a similar graph structure, and the corresponding MH nodes have the same transformation and method type. For example, there are three MHGs in Figure 4.1 on Page 69 and only two of them, $G_{N_0}$ and $G_{N_0'}$, are equivalent, as both have the same structure.

5.4.1 MH Key and MHG Key

An MH key, $\text{MH}_\text{key}_{\text{mh}}$, is an MH’s unique identifier. It is made up of an MH’s transformation name, a method type that the transformation applies
Figure 5.2: MH keys and MHG key
to, and optional data required for the transformation. The transformation name characterizes what kinds of transformation the MH has, and it is the API that creates the MH, e.g., `guardWithTest`, `insertArguments`, `filterReturn`, if it is not leaf MH. The optional parameters are only necessary for some special transformations. For example, the `insertArguments` transformation requires the existence of two variables: `pos`, the position where the insert occurs, and `values`, an array that indicates what to insert. For leaf MHs, e.g., `DirectHandle` that does not have any transformation, its MH’s transformation is `null`.

Different from MH key, an MHG key, `MHG_key_mh`, uniquely identifies the transformation that the whole MHG performs. An `MHG_key_mh` is associated with the root of an MHG, `mh`, and is made up of MH keys of all MHs in the graph (as shown in Figure 5.2). In this chapter, the MHG key of $G_{mh}(V, E)$ is defined as

$$
MHG_{key_mh} = \left\{ \begin{array}{l}
MH_{key_mh}, \text{if } mh \text{ is a leaf MH} \\
\{MH_{key_mh}, S_{mh}\}
\end{array} \right. \quad (5.1)
$$

where $S_{mh}$ is $mh$’s ordered set of its child MHs, where child order is customized by $mh$’s transformation. For example, `guardWithTest` has three children and its $S_{mh}$ is a list `[guard, target, fallback]`, while $S_{mh}$ of `filterReturn` is `[next, filter]` instead of `[filter, next]`. Corresponding to Equation 5.1, a sample MHG key sample for a `GuardWithTestHandle` is shown in Listing 5.1.

To create this MHG key, the `orderedChild` map will be filled up with child
MHG keys for guard, target, and fallback, respectively.
Thus, an MHG can be represented by its MHG key, which is recursively built from MH keys and the structure of the graph. Two MHGs are equivalent if both MHG keys are equivalent, while the equivalency of two MH keys does not mean that the graphs starting from both are equivalent.

5.4.2 MHG Equivalency Model

MHGs’ equivalency comparison is achieved by the comparison of their MHG keys. The equivalency indicates that both the graph structure and the MH nodes in the graph are completely equivalent.

The equivalency model is represented by the equivalency function $F : (m, n) \rightarrow \{true, false\}$, where $m$ and $n$ are the roots of MHG $G_m$ and $G_n$, respectively. If $F(m, n) = true$, then the MHGs $G_m$ and $G_n$ are also equivalent. The function $F(m, n)$ is defined

$$F(m, n) = (m = n) \lor (f'(m, n) \land f''(m) = f''(n)) \quad (5.2)$$
where \( f''(m) = \text{MH}_\text{key}_m \). The function \( f'(m, n) \) compares \( S_m \) and \( S_n \), of the MHs \( m \) and \( n \). Together with MH key, a simplified \( f'(m, n) \) is shown in Equation 5.3.

\[
f'(m, n) = \begin{cases} 
\left| S_m \right| \land \bigwedge_{i=1}^{\left| S_m \right|} F(S_m(i), S_n(i)), & \text{if } S_m \neq \emptyset \land S_n \neq \emptyset \land \left| S_m \right| = \left| S_n \right| \\
\text{true}, & S_m = S_n = \emptyset \\
\text{false}, & \text{Otherwise}
\end{cases}
\] (5.3)

The complexity to determine two MHG \( G_m \) and \( G_n \) is \( O(\max(\text{size}(G_m), \text{size}(G_n))) \).

For two equivalent MHGs, the detection requires one graph traversal. Based on this model, the complexity to find out all equivalent subgraphs from two \( n \)-size graphs

\[
T(n) = O(n^2) \ast f \ast O(n - 1) \approx O(n^3)
\] (5.4)

where \( f \) is the maximal number of children a node has in the graph.

### 5.5 System Design

Method Handle Deduplication System (MHDeS) is a prototype designed to deduplicate equivalent method handle graphs at a program run-time. It implements the deduplication model 5.2 from the engineering perspective and improves deduplication efficiency by adding

- a transformation index for quickly identifying MHs having a specified
transformation in the system;

- an MHG index key, representing a method handle about to be created, for comparison; and

- a fast-path comparison to avoid unnecessary comparisons by detecting non-equivalent MHs as early as possible.

Components in MHDeS are an MH pool, a purger thread (purger), a configuration, a detector, and filters, as shown in Figure 5.3. The MH pool organizes all detected unique method handle references at runtime; The purger thread periodically removes MH references that are not frequently hit during comparison; The detector compares the equivalency between an MHG in the pool and an MHG index key of an MH about to be created.
5.5.1 Method Handle Pool

A method handle is wrapped as \textit{MHObject} \footnote{In the remaining chapter, the terms: MH and \textit{MHObject}, are exchangeable, although they are different.}, which consists of an MH reference and an integer \textit{count}, which indicates the number of times the corresponding MH has been matched during equivalency detection.

The MH pool is made up of a transformation index, and MHObject chains, as shown in Figure 5.4. In the pool, all detected unique MHs having the same transformation type are chained and sorted by their \textit{counts} in descending order. Similar to the MHObject, each chain is labeled with \textit{maxCount}, which indicates the maximum \textit{count} value that MHObjects on the chain have. An MHObject chain is updated when a new unique MHObject is created and inserted, or it is purged.

![Figure 5.4: MH Pool Overview](image-url)
Transformation Index  Individual MHObject chains are indexed by transformation names. As terminal MHObjects do not have any transformations, their chains are indexed by the corresponding class names (e.g., DirectHandle and VirtualHandle). As shown in Figure 5.4, the chain with 10 MHObjects of InsertArguments is indexed by InsertArgument. With the transformation index, an MHObject chain can be retrieved in one hash table lookup, and a method handle or MHG index key is only compared to MHs in the MHObject chain that have the same transformation type as the given MH. The design of both transformation index and transformation chains is driven to reduce equivalency comparison space. With the transformation index, a method handle about to be created or an MH index key is only compared to method handles having the same transformation, and the expense to locate those method handles can be as little as $O(1)$ (one hash table lookup).

5.5.2 Detector

The detector conducts the equivalency comparison task in the MHDeS. It accepts a method handle or an MHG index key, a structure representing a method handle about to be created, and returns whether it is unique or not. The detector improves the equivalency model 5.2 from the engineering perspectives. Based on the MH pool, the main points in the detector are MHG Index Key and Fast-Path comparison.
The motivation for the MHG index key is to avoid wasting CPU and memory resources on non-unique MHs. Equation 5.2 requires the existence of two MHGs for comparison. The traditional deduplication process that creates an MHG and then has it garbage collected after identifying it is a duplication, is not efficient, because an MHG creation involves a number of memory allocations and type checkings.

An MHG index key is a data object that holds all arguments for a new concrete MH construction, and it can completely represent the MH that can be created from this key.\(^2\) This index key is made up of a transformation \((cls)\), an MH key \((mhKey)\), and its ordered child MHs \((children)\). For example, the \(children\) in an MHG index key of \(GuardWithTest\) are \([guard, trueTarget, falseTarget]\), while some other necessary arguments for transformation, such as \([pos, objects[]]\) for \(insertArgument\) transformation, have been embedded in the \(mhKey\) (as shown in Table 5.1). Terminal MHs (leaves) don’t have any MHG index key since they do not have any transformation.

\(^2\)Since an MHG can be identified by its root, an MHG index key represents an MH or an MHG that is about to be created.
5.5.2.2 Detection Procedure

Based on the MHG index key, the new MH equivalency function becomes

\[ F(m, n) = (m = n) \lor (f'(iKey, n) \land f''(iKey) == f''(n)) \] (5.5)

and

\[ f'(iKey, n) = \begin{cases} \bigwedge_{i=1}^{\left|S_n\right|} F(S_{iKey}(i), S_n(i)) & |S_{iKey}| = |S_n| \\ false & f''(iKey) \neq f''(n) \end{cases} \] (5.6)

where the \(iKey\) is the MHG index key of the MH, \(m\), to be created, and \(n\) is an existing MH with the same transformation in the pool. This comparison is recursive. Therefore, the complete detection procedure is shown in Algorithm 2.

The idea of the Algorithm 2 is that MHDeS creates an MHG index key for an MH about to be created, and compares this key to the existing MHs, which have the same transformation name, in the MH pool. The MH is created only if the detection reports the corresponding MHG index key is not equivalent to any MH in the pool.

5.5.2.3 Fast-Path vs. Slow-path

There is a fast-path comparison and a slow-path comparison for the condition \(tested.equals(mh)\) in Algorithm 2. One case of fast-path comparison is that both \(tested\) and \(mh\) refer to the same MH instance. In other words, the \(mh\) that is in an ordered child set of an index key is hit from the pool, and
Algorithm 2 Equivalency Detection and Elimination

1: procedure getUnique(cls, mhKey, children, args)
2:   indexkey = MHGIndexKey.create(cls, mhKey, children, args);
3:   mhList = MH Pool.get(cls)
4:   for all MHObeject mho: mhList do
5:     key = mho.getMHKey()
6:     if !key.equal(indexkey.getMHKey()) then ▶ fast-path $f''$
7:       Continue ▶ slow-path $f'$ now
8:     end if
9:     i = 0
10:    while i < mho.getChildren().size() do
11:       tested = mho.getChild(i)
12:       mh = get $i^{th}$ child of indexKey
13:       if !tested.equals(mh) then
14:         break
15:       end if
16:       i++
17:     end while
18:    if i < mho.getChildren().size() then
19:       mho.incr() return mho.getMH()
20:  end if
21: end for
22: mh = cls.newInstance(args, MHs)
23: mh.cacheMHIndexKey(indexKey) ▶ cache the indexKey
24: insert mh to proper pos in mhList
25: return mh
26: end procedure
there is no further comparison in this case. Another case for the fast-path comparison is when the first case fails, both \textit{tested} and \textit{mh} have unequal MH keys and comparison result is returned directly. By fast-path comparison, the deduplication can make some fast assertions whether \textit{tested} and \textit{mh} are equivalent or not.

Slow-path comparison, on the other hand, only compares the graph keys of both \textit{mh} and \textit{tested} recursively, by following Equation 5.2. The slow-path will only be conducted, if fast-path comparison could not make an accurate assertion (i.e., both are not the same instance, and they have equal MH keys). Slow-path comparison is time-consuming, while the fast-path comparison is an early detection of un-matched MHs to avoid the cost of slow-path comparison.

The overhead for the operation \texttt{tested.equals(mh)} is largely reduced by the MH pool. This is because most MHs in the \textit{children} of an index key are unique, and have been placed in the pool. Thus, a comparison result can be quickly made by testing whether both are the same instance during the fast-path comparison.

\subsection*{5.5.3 Purge}

MHDeS periodically checks the MH pool, removing MHO Objects if they are rarely matched during equivalency detection. In the pool, two terms, \textit{count} and \textit{maxCount} that is maximal \textit{count} in a transformation chain, are used to indicate the frequency of an MHO Object and transformation chain.
There are two kinds of purge operations \textit{MHObject purge} and \textit{Transformation chain purge}. The former removes MHObjects, the \textit{counts} of which are less than configured threshold, and the latter removes the whole chain if the \textit{maxCount} is less than another configured threshold. Along with the purge, the chain is re-sorted again by MHObject’s \textit{count}. Instead of sorting it during detection, the re-sort operation following purge aims to reduce potential waiting time caused by MHDs.

Both purges are conducted in the purge thread at a fixed rate. Normally, the chain purge occurs with a longer interval than that of MHObject purge since the chain purge requires a synchronization on the transformation index, so that fewer contentions will be triggered between the chain purge thread and the main thread. For simplification, the default intervals for both purges are set to one second. This is because MHG creations only happen in early execution phase, instead of a later phase, since most of MHGs after deduplication are likely to be JITted, and their native code are inlined at dynamic call sites directly in a later phase. Correspondingly, intervals for both purges should be large enough, so that the cache-hit number for MHs in the pool can be maximized at the beginning, and all these MHs are cleared from the pool in the later execution phase.

5.5.4 Filter

This model conducts transformation filtering from MHDs to balance the deduplication overhead and the potential benefits. According to the first
finding in Section 4.4, equivalent MH ratios of different kinds of method handles—the number of equivalent method handle pairs to the total number of equivalent method handle pairs—vary sharply. Thus, it is a bonus to filter those method handles, the transformations of which have trivial equivalency ratio \[103\]. In our evaluation, two transformations (i.e., \textit{FilterReturnHandle} and \textit{ExplicitCastHandle}) are configured.

5.6 Evaluation

In this section, the CLBG JRuby micro-indy benchmark is scheduled on the J9 JVM with MHDeS. The MHDeS is evaluated in terms of execution time, memory, and JIT compilations.

5.6.1 Execution Time Impacts

Figure 5.5 shows our comparison results and performance improvements made by MHDeS, respectively. In these figures, the bars labeled with \textit{MHDeS} represent the data with MHDeS, while the one with \textit{Orig} represent the runtime without MHDeS. In Figure 5.5a 5.5b, the y axis is improved performance \(ip\), which is evaluated as

\[
\frac{\text{ExecTime}_{\text{Orig}} - \text{ExecTime}_{\text{MHDeS}}}{\text{ExecTime}_{\text{Orig}}} \times 100\% 
\]

(5.7)

According to Figure 5.5, MHDeS speeds up benchmark’s execution on av-
Figure 5.5: Execution Time Performance Comparison

(a) Filter Off + ExecTime ip (Median: 0.9%, Mean: 4.67%, std: 14.48)

(b) Filter On + ExecTime ip (Median: 1.55%, Mean: 2.77%, std: 4.84)
verage. In the figure, the average speedups are 4.67% and 2.77% when filter is off and on, respectively. Although these average improvements are only moderate, the improvements of the deduplication on tests vary. For example, the maximal execution speedup can reach as high as 77%, and the minimal speedup is only -6% when filter is off. This dramatic change in improvement is caused by different MHG constructors among individual tests. Different tests have their preferences for MHG transformations, and the overhead to interpret and deduplicate these MHG transformations also vary. For example, there is rarely any MHG transformation in test `eval` and `so_sieve`, while there are more than 8 kinds of transformations in the test `so_matrix`.

The transformation for filtering has significant impact on the elapsed CPU time measure, and these filtered transformations should be carefully chosen to maximize performance speedup. First, the deviation of ip is reduced from 14.48 to 4.85 when filter is enabled, as shown in Figure 5.5b. This makes the distribution of ip among different tests smoother. Second, the MH transformation for filter should be carefully chosen. In Figure 5.5b, the ip for test `so_matrix` is reduced from 77% to 8%, while it increases from 0.5% to 25% for test `spectral_norm` when filter is enabled. This change in the ip can be explained as the trade-off between the overhead of the filtered MH transformation deduplication and the benefits of avoiding these equivalent MHG creations. However, the relation between ip and filtered transformation is not determined from both figures, because an MHG starting with a filtered transformation can have a number of MHs of the other non-filtered
transformation types.

5.6.2 Memory Impacts

Three measures are used to indicate MHDs memory performance: memory usage (the amount of memory occupied after the last GC), the number of global Garbage Collections (GCs), and the total GC pause time. Each measure is affected by various factors. Take the memory usage as an example, MHDs, on one hand, reduces memory consumption by eliminating equivalent MHG creations. On the other hand, it increases the amount of used memory as it repeatedly creates extra temporary objects, e.g., MHObject, MHG index key, etc, during equivalency detection and holds them in the pool.

The GC policy used in our experiment is the default gencon in J9, which aims to maximize throughput. With this policy, a heap is divided into a nursery and a tenured area, and objects are first created in the nursery and promoted to the tenured area if these objects survive a certain number of GCs.

Memory usage Figure 5.6a shows the memory usage reduction for tests in the CLBG benchmark. In the figure, the data for test list is dirty data as the test execution fails with stack over flow exception. For test mbari_bogus1 that has negative memory usage reduction, the equivalent ratio is not high, and most of MHs in the pool are cleared, when the program is about to
Figure 5.6: MHDeS impacts memory usage and Number of GCs, Filter off.
Figure 5.7: GC Pause Time impact, Filter Off (Mean: 1.65%)
According to the figure 5.6a, MHDeS can reduce memory usage by 7.19% (list is excluded) on average. For those benchmarks that have negative reduction percentages, the memory occupied by MHDeS (mainly MHO objects and chains) outweighs the reduced equivalent method handles. This number is less than mean SMR 28.83% that was discussed in Section 4.4, because the occupied memory contains other non-MH data structures.

**Number of global GCs** The comparison for the number of GCs is shown in Figure 5.6b. Accordingly to the figure, the number of global GCs is reduced by 0.26% when the filter is off. The conclusions for this are 1) overall, MHDeS does not improve program runtime in terms of the number of GCs; and 2) MHDeS helps the reduction of memory usage at runtime. These conclusions can be explained by the fact that the size for newly created objects (the one in the MH pool and the one for the deduplication) is less than that of reduced equivalent MHGs.

**GC Pause Time** The percentage in reduced GC pause time $\gamma$ for a test is shown in Figure 5.7. According to the figure, the GC pause time is reduced by 1.65% on average, which is only a minor improvement. With MHDeS, there is an increase in short-lived objects, i.e., temporary objects, and de-
crease in the number of MHGs. In this case, the incremental GC overhead on temporary objects is largely counteracted by the reduced GC overhead of the equivalent MHGs. Besides, MHs are only a small portion of objects created by the bytecode compiled from those tests at runtime.

5.6.3 MH Pool Purge Interval vs. Performance

Two tests, *cal* and *app_factorial*[^3] are selected for repeated execution with varied MHDeS pool purge intervals. Figure 5.8 illustrates the relationship between purge interval and execution time. Based on the figure, the execution time for both tests first decreases, and then increases until it becomes stable.

[^3]: Both are single thread applications.
In other words, there is an optimal purge interval value for each single test. The explanation relies on the MH creation behaviors and the MH pool itself. First, MHGs are only created in an execution’s early phase, during which dynamic methods are warmed up. In the later phase, many of these created MHGs become hot and inlined to method call sites directly, instead of creating new one. In other words, MHGs cached in the MH pool are not likely to be used in the later execution phase, and they should be cleared from the pool. Second, a large purge interval value would keep more unique MHs in the pool, for the benefit of avoiding creating equivalent MHs. However, this also implies more comparison overhead during detection, as MHDeS continually iterates MH chains until it finds a matched one. Thus, there would be an optimal interval value for benchmarks, depending on MH creation patterns that these benchmarks have.

### 5.6.4 JIT Compilation Impact

The number of MH JIT compilations remains stable regardless of MHDeS, as shown in Figure 5.9. In J9 JVM, the default JIT compiler is a method-based compiler TR[61]. According to the figure, the number of JIT compilations for the test so_matrix is reduced from 75 to 55, which is the maximal change of JIT compilation. For other tests, there is trivial change in the number of JITs.

Based on the existing conclusion that the reduction of the ExecTime and trivial paused GC time, MHDeS helps move MH JIT compilation earlier. First,
Figure 5.9: Impacts on the Number of MH JITs, Filter Off
the JIT compilation is conducted asynchronously. Second, Equation 3.1 can be simplified to

\[ \text{ExecTime} \approx \sum_{i=1}^{n_1} T_{\text{int}_i} + \sum_{i=1}^{n_2} T_{\text{exe}_i} \] (5.9)

as the percentage of accumulated paused GC time to the total number of \text{ExecTime} is trivial. Since the bytecode method for JIT is nearly fixed (i.e., tests in the JRuby benchmark repeat long loops), the only explanation for the reduction of \text{ExecTime} (4.67\%) is earlier JIT compilation. Otherwise, there would be reduced \( \sum_{i=1}^{n_2} T_{\text{exe}_i} \), but an increase in \( \sum_{i=1}^{n_1} T_{\text{int}_i} + \sum_{i=1}^{n_2} T_{\text{exe}_i} \), which would result in a contradiction.

### 5.6.5 Deduplication Cost and MHG Reduction Efficiency

MHG Reduction Productivity (MRP) in Equation 5.10 is a ratio of the number of an MHG index key \((n_0)\), which is detected to be equivalent, to the total number of MHGs index keys \((n)\), which MHDeS receives for equivalency comparison. This measure indicates that MHDeS’s productivity, and the higher MRP is, the more productivity MHDeS is.

\[ \text{MRP} = \frac{n_0}{n} \times 100\% \] (5.10)

Figure 5.10 shows the MHG reduction productivity when filtering is disabled. In the figure, we can see clearly that as much as 32\% of method handles
Figure 5.10: MHG Reduction Productive, Filter Off, Mean: 32%
(or MH index keys) are detected to be equivalent and eliminated at JRuby benchmark runtime. For some tests, the MRP can reach as high as 53%, which proves that MHDeS helps in the reduction of equivalent MHs.

The MHDeS expense is a time expense for an MHG index key equivalency detection. As shown in Table 5.2, the average expense is trivial, as the mean maximal time expense for individual tests is about 10ms (after excluding the dirty data for test list, and the maximal expense is 167ms for mbari_bogus1, the ExecTime of which is about 26.04s). Besides, the mean deduplication expense is less than 1ms because the expenses for the majority of deduplication are nearly 0 as the tested and mh in Algorithm 2 refer to the same MH in the MH pool.

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Mean(ms)</th>
<th>Max(ms)</th>
<th>Test Name</th>
<th>Mean(ms)</th>
<th>Max(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>app_pentomino</td>
<td>0.163418</td>
<td>10</td>
<td>monte_carlo_pi</td>
<td>0.415842</td>
<td>11</td>
</tr>
<tr>
<td>so_array</td>
<td>0.206452</td>
<td>10</td>
<td>gc_string</td>
<td>0.375</td>
<td>15</td>
</tr>
<tr>
<td>count_multi threaded</td>
<td>0.171504</td>
<td>11</td>
<td>app_tarai</td>
<td>0.241667</td>
<td>10</td>
</tr>
<tr>
<td>count_shared_thread</td>
<td>0.180247</td>
<td>15</td>
<td>so_object</td>
<td>0.179426</td>
<td>9</td>
</tr>
<tr>
<td>observ</td>
<td>0.195558</td>
<td>11</td>
<td>nsieve_bits</td>
<td>0.180233</td>
<td>8</td>
</tr>
<tr>
<td>eval</td>
<td>0.468085</td>
<td>9</td>
<td>write_large</td>
<td>0.40762</td>
<td>7</td>
</tr>
<tr>
<td>string_concat</td>
<td>0.220039</td>
<td>8</td>
<td>mergesort_hongli</td>
<td>0.222973</td>
<td>9</td>
</tr>
<tr>
<td>nbody</td>
<td>0.179502</td>
<td>9</td>
<td>fiber_ring</td>
<td>0.26898</td>
<td>9</td>
</tr>
<tr>
<td>ge_array</td>
<td>0.2</td>
<td>9</td>
<td>printf</td>
<td>0.33739</td>
<td>8</td>
</tr>
<tr>
<td>partial_sums</td>
<td>0.281425</td>
<td>10</td>
<td>fasta</td>
<td>0.23214</td>
<td>9</td>
</tr>
<tr>
<td>word_anagrams</td>
<td>0.197044</td>
<td>9</td>
<td>list</td>
<td>0.94506</td>
<td>296</td>
</tr>
<tr>
<td>gc_mib</td>
<td>0.285714</td>
<td>8</td>
<td>simple_connect</td>
<td>0.223214</td>
<td>12</td>
</tr>
<tr>
<td>binary_trees</td>
<td>0.297872</td>
<td>10</td>
<td>cal</td>
<td>0.181435</td>
<td>8</td>
</tr>
<tr>
<td>app_factorial</td>
<td>0.209302</td>
<td>6</td>
<td>app_fib</td>
<td>0.211009</td>
<td>8</td>
</tr>
<tr>
<td>so_lists_small</td>
<td>0.190311</td>
<td>11</td>
<td>so_matrix</td>
<td>0.378261</td>
<td>9</td>
</tr>
<tr>
<td>so_sieve</td>
<td>0.272727</td>
<td>8</td>
<td>pi</td>
<td>0.178571</td>
<td>9</td>
</tr>
<tr>
<td>so_lists</td>
<td>0.273565</td>
<td>7</td>
<td>app_mandelbrot</td>
<td>0.376</td>
<td>10</td>
</tr>
<tr>
<td>primes</td>
<td>0.252137</td>
<td>10</td>
<td>socket_transfer_1mb_noblock</td>
<td>0.170673</td>
<td>1</td>
</tr>
<tr>
<td>spectral_norm</td>
<td>0.194774</td>
<td>9</td>
<td>fractal</td>
<td>0.231884</td>
<td>10</td>
</tr>
<tr>
<td>simple_server</td>
<td>0.201681</td>
<td>9</td>
<td>socket_transfer_1mb</td>
<td>0.204334</td>
<td>1</td>
</tr>
<tr>
<td>mbari_bogus1</td>
<td>0.0388703</td>
<td>167</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: MHDeS Expense
Chapter 6

GraphJIT and Dynamic Bytecode Generation for Method Handle Graphs

This chapter provides a dynamic bytecode generation (i.e., dynamic method handle graph compilation, or graph inlining) solution to simplify MHGs. The chapter is organized as follows. Section 6.1, Section 6.2, and Section 6.3 provide the motivation, goals, and related work for dynamic MHG compilation. Section 6.4 and Section 6.5 present an overview of MHG compilation, and GraphJIT’s concept and its design. Section 6.6 describes a method handle template system and demonstrates how it is integrated with GraphJIT and the JIT compiler in J9. Section 6.8 discusses extended caches (i.e., the object cache and the class cache) that are implemented in JSR292 package for
GraphJIT. Section 6.9 evaluates the MHG bytecode generation, and presents results from the perspectives of execution time, memory garbage collection, and JIT compilation.

In this chapter, the terms graph transformation, graph fusion, graph simplification, and graph compilation are interchangeable.

### 6.1 Motivation

For method handle graphs, there are two kinds of traversals: the method-based graph traversal and the object-based graph traversal. The method-based graph traversal refers to a traversal that walks through method calls on graph nodes, while the object-based graph traversal walks through following references between graph nodes. For example, the execution of an MHG is a method-based graph, while marking live graph nodes during GC is an object-based graph traversal, in which a node is transferred to another node by dereferences.

**Graph Traversal** Large graphs challenge programming languages’ runtime, owing to their complex structures for traversal. At runtime, a graph node is mapped to a memory range, and an internal node of a graph serves as an indirection, connecting two non-adjacent nodes. The number of these indirections that an interpreter or executor has to visit from the root to leaves slow down the execution, and the instantiating of these internal nodes also
increases memory consumption and garbage collection overhead, especially when the graph size (both nodes and edges) becomes larger. Take a simple object graph in Figure 6.1 for example, one field of object 1 references object 3, which in turn holds references to three other objects. To reach leaf object 7, the runtime has to visit at least three internal nodes (i.e., 3, 4, and 6) from the root object 1, regardless of the graph traversal algorithm (e.g., depth-first-traversal or breadth-first traversal). For a graph with \( V \) nodes and \( E \) edges, its traversal cost, approximately \( O(V + E) \) \(^{[43]}\), becomes predominant, and the repeated traversals of a graph are unnecessary.

An MHG is a special kind of graph, and the main operation on an MHG is a method-based traversal that transfers a dynamic method invocation at a call site to a number of real method implementations. During traversal, an MH's execution recursively triggers the executions of its children. Owing to the prevalence of dynamic method invocations in a dynamically typed language, it is worth investigating graph traversal optimizations from runtime perspectives.

**Just-In-Time Compilation** Large complex MHGs are also inefficient for Just-In-Time (JIT) compilations on the JVM. In J9, JIT threads are shared among for all compilation tasks, and the large number of MH compilation tasks would increase competitions among all tasks (e.g., MH compilation tasks and non-MH compilation tasks) for JIT threads. Besides, profiling is not free, and each JIT compilation has a start-up cost. The accumulation of
Figure 6.1: Graph for Traversal

(a) merge 1 and 3

(b) Merge 3, 4, and 6

Figure 6.2: Two Transformations

(a) merge 1 and 3

(b) Merge 3, 4, and 6
these costs is approximately proportional to the number of MHs in a graph.

**Garbage Collection** It is still indispensable to traverse the original MHG graph even if the MHG has been JITted. Although a JIT compilation (e.g., method based JIT compilation) benefits the execution of method handle graphs by compiling their bytecode methods to native code, it does not help MHG GC\(^1\) especially for nodes that are far away from the root of the graph. This is because a graph’s node is represented as a region in memory, and the whole graph is made up of these regions via linking (i.e., object reference). For a node’s method, a JIT compilation only translates its bytecodes into machine code without any changes to the graph structure. Thus, the system has to walk through the original graph to mark all live graph nodes during GC, regardless of JIT compilation.

**Transformation Pattern and Other Optimizations** Based on the first MHG mining’s finding in Section 4.4, a small number of transformation patterns occur much more frequently than others. The JVM fails to take advantage of the mining results, and does not distinguish MHs of these frequent patterns from other infrequent ones. As a result, these frequent transformations are treated the same as other infrequently ones, and no optimization is done until they are JITted.

\(^1\)In this dissertation, garbage collection refers to the mark-sweep based algorithms if not specified.
6.2 Goal

Addressing the issues listed in Section 6.1, this chapter provides a dynamic method handle graph transformation solution. This solution translates an object graph (i.e., method handle graph) of frequent transformation patterns into another equivalent but simpler graph (fewer nodes and edges), and substitutes it with the simpler graph for the execution. Therefore, a new layer between the bytecodes produced by dynamic JVM language interpreters and the native JIT compilers in the JVM, is provided, as shown in Figure 6.3. This new layer takes MHGs produced by dynamic JVM language interpreters as input, and outputs simpler ones, after performing optimizations.

The core technique in the solution is a dynamic bytecode generation from bytecode to bytecode in such a way that a) a graph’s leaf objects are moved close to the graph’s entries (i.e., root nodes), and b) hot internal graph nodes

---

2The terms *Dynamic Bytecode Generation*, *MHG Inlining* and *Graph Inlining* are interchangeable in this dissertation.
are merged into a single one if possible. For example, two transformations in Figure 6.2 are provided for the graph in Figure 6.1: two nodes, 1 and 3, are merged into a single (1, 3) in Figure 6.2a, while three internal nodes, 3, 4, and 6, are merged together in Figure 6.2b. In Figure 6.2b, object (3,4,6) is newly created, and the original objects, 3, 4, and 6, are no longer held by the graph and might be for for the next GC. Intuitively, the traversal efficiency of both the newly generated graphs would beat that of the original one, considering the number of nodes in the graph.

The goals of a method handle graph simplification are to

- reduce graph traversal cost. After transformation, the graph has fewer nodes. As each node represents a method, there would be fewer indirections during method invocations (i.e., the number of graph nodes).

- reduce potential Just-In-Time (JIT) compilation overheads on graphs (i.e., method handle graphs). By the graph transformation, the number of graph nodes is reduced, so that the JIT compilation overhead can be reduced because of a) fewer profiling targets; b) fewer method handle nodes for potential JIT compilation.

### 6.2.1 Comparison

**Dynamic MHG Inlining vs. Handcrafted Inlining** Based on the transformation pattern mining results, a small number of transformations

---

3The operation of object merging in this chapter refers to the generation of a class that fuses classes of these objects and the new class.
occur much more frequently than others. Thus, a straightforward idea is handcrafted inlining that creates a new combined transformation (i.e., a new MH class) to represent a frequent transformation chain. The advantage of this approach is that there is little runtime overhead, as the handcrafted transformation is added to the Java standard package at beginning. The main problem for this solution is that the found frequent transformation patterns from CLBG JRuby micro-indy benchmark might not be universal for all dynamic JVM language interpreters (e.g., Nashorn and Jython) or other benchmarks of the same interpreter. It would be a source of performance loss for the JVM, if transformation patterns among these interpreters are not consistent with each other.

Dynamic MHG inlining has merits. Compared to the handcrafted inlining, it is flexible and is not constrained to a specified interpreter or benchmark. This mechanism would work for all dynamic JVM interpreters, as it automatically selects frequent transformation sub-chains for the fusion. However, its disadvantage is its cost at runtime.

Selective Inlining vs. Unselective Inlining For the Unselective inlining, all MHs in a graph are fused together, while only a small number of hot MHs are selected for fusion in selective inlining. The decision to do selective inlining is motivated by the inlining cost and MH hotness. For a single MH, the inlining cost can be expenses, such as bytecode parsing, code generation, and verification. Costs vary among MHs of different transformations and
graph structures. For example, an MH that performs a complex transformation is likely to result in large complex bytecodes, which would require more time for analysis. Similarly, each MH has its hotness during interpretation, and a hot MH is likely to be executed again in the future, while the infrequently executed MHs are vice versa. Therefore, it is not economical to apply inlining to these infrequently executed MHs, as they are not likely to be executed frequently in the future.

6.3 Related Work

The related topics to dynamic MHG compilation are object inlining, MH inlining, object co-location and inline caching.

6.3.1 Object Inlining

Graph inlining (i.e., Object inlining) transforms heap data structures by fusing parent and child data together, and this idea was first proposed by Dolby and Chien \[48, 47, 46\] for a dialect of C++. In their solution, a field is classified to be inlinable if it satisfies data-flow criteria and one-to-one relation (i.e., every parent corresponds to exact one child during execution). Although their evaluation shows that the object inlining can result in 14% performance speedup, the solution strongly depends on field patterns and static analysis, while the runtime information is ignored.

Based on Dolby’s work and execution traces of Java benchmarks, Lhotak
and Hendren [70] categorize fields with inlinable patterns (i.e., one-to-one, field-specific one-to-one, and unique-store), and non-inlinable patterns, and then analyzes the performance impact, if inlining is applied to all fields with inlinable patterns. One conclusion is that opportunities for inlining and performance impacts are strongly dependent on the benchmarks (e.g., the categorization of inlinable fields), and are quite limited for many benchmarks. This conclusion is similar to our evaluation result.

6.3.2 Method Handle Inlining

As a key component of the JSR 292, method handles are important candidates for JVM optimization. Thalinger and Rose from Oracle introduce an initial method handle chain inlining in HotSpot JVM [93], which is comprised of a walker and a method handle compiler. The walker presents bytecodes to the method handle compiler, which compiles the whole chain into a stand-alone bytecode method, method handle adapter. First, this inlining mechanism is a part of MH JIT compilation in HotSpot, and its purpose is only to provide adapters for the server compiler, instead of the interpreter, while graph inlining in this dissertation is for both interpretation (the source MHG is replaced by the generated one) and MH JIT compilation. Second, bytecodes from walker are used to generate a single adapter without any selection, while graph inlining is dynamic, and only a subset of nodes are selected for the fusion. Third, the graph inlining is independent from the JVM platform, and is configurable for all other similar graphs. GraphJIT,
together with the template system in J9 JVM, processes MHGs on bytecode level, for the benefit of both MHG JIT compilation (i.e., fewer MHG JIT tasks) and interpretation (i.e., fewer indirections from root to leaves).

6.3.3 Co-Location

Wimmer et al. provide another kind of object fusion (i.e., object co-location) for the HotSpot JVM without object layout modifications. In their solution, objects that are accessed together are placed next to each other in the memory, allowing field loads to be replaced by address arithmetic [100, 99, 102]. In their solution, they first use read barriers to detect the most frequently accessed fields. Then garbage collection is modified to co-locate these frequently accessed objects that have parent-child relationships: it copies them to a consecutive region as a group. To make object co-location effectively, the client JIT compiler removes field loading instructions, when parent-child satisfies some pre-conditions. Different from Wimmer’s work, Shuf et al. provide co-allocation and GC time co-location based on the concept of prolific types, objects of which tend to have short lifetimes and suitable for GC [91]. During creation of an object of prolific type, the co-allocation reserves enough memory space right next to the object for its children of prolific types. During the GC, the graph traversal is modified, so that objects of prolific types in a region are given a high priority, as these objects might have high locality and yield less traversal delays. Similar to Shuf’s co-allocation, Clifford et al. introduce allocation folding [42], which groups allocations of multiple sites in
an optimized function into a single larger allocation group, for V8 JavaScript engine. The differences between co-location and GraphJIT are

- **graph structure.** In Wimmer’s solution, the relationship (i.e., field) between a parent and a child cannot be overridden once both are created. GraphJIT operates on Frequently Traversed but Stable Directed Acyclic (FTSDA) graphs (e.g., MHG), and it handles graph’s mutations by inline caching optimization.

- **object layout.** GraphJIT dynamically emits bytecode classes and creates a new graph to replace the source graph for execution. This means the object layout is modified at program runtime, while all these related co-location solutions do not involve any layout modification.

- **Independence.** GraphJIT is an independent library for FTSDA graph simplification, and it works on the bytecode level. In other words, it is independent from the runtime system (e.g., memory management and JIT system).

Another relevant but different optimization is *object combining* [97], which combines objects that have similar lifetime in the same place at allocation time, so that both strain on the memory management and the number of pointer indirections during a program’s execution can be reduced. This idea is more aggressive than the Shuf’s co-allocation, as it supports combination of objects, even these objects are not related. In Shuf’s system, the decision to make objects that have similar lifetime, is based on the profiling information.
that counts invocations of individual allocation sites, plus compiler’s dataflow analysis. Although this solution is reported to have 20% to 34% speedup for some tests, the expense is relatively high, if objects for combination do not conform to the assumption.

6.3.4 Inline Caching

Inline caching (Section 2.4.1.4) optimization in GraphJIT is a well-known compilation optimization. This optimization was initially proposed for the Self interpreter [65, 66] to remember virtual method invocation receivers that were resolved previously. The solution in this dissertation adopts IC to handle mutable graph nodes, by generating fields to remember receivers during runtime. As a method handle graph is relative stable, IC is only used when GraphJIT is forced to inline mutable nodes from users.

6.4 Overview

The idea of method handle graph compilation in this chapter is implemented as a Graph Just-in-Time (GraphJIT) compiler and a method handle template system. GraphJIT is a graph compiler that translates a FTSDA graph into an equivalent but simpler one, while the template system is a bytecode generation system that provides MH bytecodes for GraphJIT.

In this chapter, we distinguish two kinds of graphs: an object graph that is made up of objects having reference relationships, and a class graph that
is built from class definitions. GraphJIT conducts object graph transformation via bytecode generation. Different from existing compilation systems, GraphJIT has attributes listed below

- dynamic bytecode generation. GraphJIT conducts bytecode translation at program runtime. For an input object graph, GraphJIT translates its bytecodes to another bytecode version. At program runtime, the object graph that is generated from the new version is expected to be simpler and more efficient for traversal than the input graph.

- graph translation. Different from other compilers, GraphJIT only targets a graph structure, and it aims to simplify graphs.

- more suitable for Frequently Traversed but Stable Directed Acyclic (FTSDA) graphs. A FTSDA graph is assumed to be stable for a period, so that the corresponding simplified graph can replace it for execution. GraphJIT detects a graph’s frequency by looking graph nodes’ invocation counters, and a node is not classified as frequently traversed, if its invocation counter is less than the compilation threshold. Similarly, a graph is recognized to be stable by analyzing the corresponding bytecode instructions, together with nodes’ type information.
6.5 GraphJIT

In this section, GraphJIT is discussed from three perspectives: concept in Section 6.5.1, its procedure in Section 6.5.2, and its design in Section 6.5.3. GraphJIT targets Frequently Traversed but Stable Directed Acyclic (FTSDA) graphs that are characterized as

- acyclic graph. GraphJIT is designed for acyclic graph simplification. GraphJIT is not able to detect a cycle. In case a cyclic graph is given, GraphJIT traverses each node, and might generate an inlined version for each node in the cycle by fusing the node with its following MHs in the graph, depending on the node it traverses and the number of times the node has been visited.

- frequently traversed graph. This means that the graph is traversed frequently, or graphs with equivalent structure are created continuously and traversed during runtime.

- stable graph. The graph structure is rarely modified once it is created.

- nodes have same type. All nodes in a graph have the same type (i.e., class, base class or interface).

6.5.1 Concepts

In this Section, two terms, class graph and object graph, are first discussed in Section 6.5.1.1 and Section 6.5.1.2, respectively. Then a graph transla-
tion example is provided in Section 6.5.1.3 and the compilation concept is provided in Section 6.5.1.4.

6.5.1.1 Class Graphs and Concrete Class Graphs

A class in an object-oriented programming language (e.g., C++ and Java) is a set of fields $S_f$ and methods $S_m$, defining the properties and behaviors that an instance of that class has. A class graph in the context of programming languages is a directed graph, consisting of nodes and edges. In a class graph, a node represents a class; an edge from a source node to a target node represents a field, the class of which is the target node.

A concrete class graph (CCG) is a specific class graph, in which all classes of nodes are non-abstract (i.e., the class of a node does not have any abstract or unimplemented interface methods). Owing to type polymorphism, a set of class definitions can build an infinite family of class graphs. For example, the CCG in Figure 6.4 can be easily deduced from class definitions in Listing 6.1.
interface INode{
    protected Object invokeExact();
}

class Node1 implements INode{
    final INode n1;
    final INode n2;
    //Methods below.
    public Node1(INode a, INode b){
        n1 = a;
        n2 = b;
    }
    @Override
    public Object invokeExact(){
        if ((boolean)n1.invokeExact()){
            return n1.invokeExact();
        } else{
            return n2.invokeExact();
        }
    }
}

class Node2 implements INode{
    final int[] arr = new int[5];
    @Override
    public Object invokeExact(){
        return arr.length;
    }
}

Node2 o1 = new Node2();
Node2 o2 = new Node2();
Node2 o3 = new Node2();
Node1 o4 = new Node1(o1, o2);
Node1 o5 = new Node1(o2, o3);
Node1 root = new Node1(o4, o5);

Listing 6.1: Class definition for Class Graph
6.5.1.2 Object Graphs

An object graph consists of objects that have direct or indirect reference relations. An object is an instance of a class, and it corresponds to a memory region at program runtime. A linking edge between two nodes in an object graph is the same as that in the corresponding class graph. For example, the root of the object graph built from Listing 6.1 has two child nodes: o4 and o5, as shown in Figure 6.5. The edges between root and its two children are labeled by n1 and n2, respectively.

The relation between an object graph and a concrete class graph can be easily determined. Intuitively, an object graph can be uniquely mapped to a concrete class graph, as types of individual objects in the object graph are non-abstract. A concrete class graph can be mapped to an infinite set of object graphs, which share the same graph structure.

Graph traversal can be treated as a method call graph [62]. For an object graph, an invocation on a graph node can be transferred to its children, which in turn propagate these invocations to further children in the graph. Via these invocations, all (or some) graph nodes are visited and this execution path
follows the edges in an object graph. The traversal can also be with iteration of a node’s children directly. For example, in mark-sweep GC, the nodes in an object graph are iterated to mark live objects by checking the fields of a node directly.

### 6.5.1.3 FTSDA Principle and Example

A Frequently Traversed but Stable Directed Acyclic (FTSDA) graph is a rooted directed graph, where all nodes are of the same type (i.e., the same super class or interface), and the graph structure is rarely modified. FTSDA graphs are prevalent in programming languages and software systems. For example, an action graph is normally created by a HTTP server application to serve incoming HTTP requests. Each action passes the request to a child action, based on a request’s headers (e.g., user-agent, accept, and cookie). Method handle graphs are also FTSDA graphs, because most of MHGs are constant, once they are created.

The idea of graph simplification in GraphJIT is to compile a FTSDA graph into another equivalent but simpler version, and then to replace the original graph with the new version for execution.

The principles to do compilation are

- merge an object graph’s internal nodes when possible, and
- move leaf nodes close to the graph’s root.

We use an example to illustrate how this simplification works. We still take
Figure 6.6: Object Graph Simplification Illustration (A dash edge means that the edge has not been processed by GraphJIT.)

the aforementioned object graph in Figure 6.1 as input and the procedure is shown in Figure 6.6. The translation order is from bottom to top. In step a), node 4 and 6 are merged into a single node (4,6) when it is visiting node 4. Next, node 3 is merged into node (4,6) in step b) to reduce one further internal node. Finally, the root node 1 is also merged to (3,4,6). After these three steps, a new object graph is generated, which has 3 less nodes and 5 less edges, and the distance from root node to the leaves 7 and 8 is shortened from 4 to 1.

6.5.1.4 GraphJIT Translation

GraphJIT is a solution to accomplish FTSDA graph simplification. GraphJIT is a JIT compiler that performs an object graph’s translation at program runtime, and this translation works at the bytecode level (i.e., both source and target are bytecode).

GraphJIT is only interested in FTSDA graphs, and an MHG is one kind
of FTSDA graphs. As a FTSDA graph is frequently traversed, GraphJIT translates the FTSDA graph into a simpler graph, and substitutes it for the original FTSDA graph for execution. This substitution works as a FTSDA graph is stable (the graph structure is rarely modified).

GraphJIT works in a similar way as shown in Figure 6.6. One difference is that GraphJIT only creates the final object \((1,3,4,6)\) but not other intermediate object nodes (i.e., \((4,6)\) and \((3,4,6)\)), for the purpose of compilation overhead reduction. Furthermore, the translation is conducted at the bytecode level, instead of object level; this will be discussed in the next section.

### 6.5.1.5 GraphJIT Overview

There are in total four main steps in GraphJIT to complete translation:

**Object-CCG graph conversion**  Object-CCG conversion generates a unique concrete class graph from an object graph. This conversion can be simply achieved by replacing nodes in an object graph with these nodes’ class names. For any class node in a CCG, its bytecode can be retrieved from corresponding class loaders.

**Dynamic bytecode generation**  This step translates the converted CCG to the target CCG. This translation is accompanied with source CCG depth-first traversal, while the target CCG generation is in the order from bottom to top. When visiting a node in the source CCG, GraphJIT first checks whether its children fields have been visited. If not, it visits its children first.
 Otherwise, it scans bytecodes of the current node, and fuses these bytecodes with the current Class’s children.

As a CCG is made up of nodes and each node represents a Java class, the bytecode generation for a CCG consists of a number of generations on a CCG’s nodes. For a class node (i.e., a node in CCG), GraphJIT scans its components (e.g., class name, fields and methods), and emits new bytecode after necessary transformations. The fusion (i.e., class renaming, field fusion, and method fusion) is triggered when GraphJIT jumps from one node to its children, and will be discussed in Section 6.5.2.

Algorithm 3 shows how bytecode generation works on an object graph node pivot. In the algorithm, GraphJIT creates an initial empty bytecode queue, class node (cn), to track all emitted bytecode instructions for each object graph node. The queue adds instructions to the tail in order and finally represents a Java class once its queued instructions are loaded into a class loader.

Sharing of the generated bytecode As a CCG can be mapped to multiple object graphs, bytecodes that are generated from one CCG can be shared for other CCGs if they are equivalent (i.e., both have equivalent structure and classes of individual graph nodes are the same). GraphJIT uses a global map, named store, in Algorithm 3 to track generated bytecodes by an object graph’s root.

GraphJIT checks in store whether the bytecode generation for the object
pivot has been done previously. If yes, the cached bytecodes (i.e., ClassNode) are directly used, instead of graph traversal for bytecode generation. Similarly, GraphJIT puts the pair of root object pivot and its cn into the store at the end of one generation.

The purpose of store is for bytecode sharing, so that the expense of visiting components of pivot’s class is eliminated directly. First, an object pivot would be hit in store directly if GraphJIT has emitted bytecodes for it before. Second, an object graph would also hit in store, if the generation on another equivalent graph has been done before. Thus, both methods hashCode and equal are overridden for FTSDA nodes, to indicate the graph structure.

**Activation of the transformed CCG**  
The activation of the transformed CCG is a process that loads a cn’s instructions, and instantiates the generated class based on the existing object graph. The new object graph is equivalent to the existing object graph, and replaces the existing object graph for the execution. For example, after object 1’s class node has been visited in Figure 6.6 GraphJIT instantiates a new object (1,3,4,6), and sets up its children using existing leaf object 2, 5, 7, 8, and 9. The new object graph starting from (1,3,4,6) then replaces the original object graph for execution.

### 6.5.2 Bytecode Generation Procedure

The bytecode generation for a class in a CCG mainly consists of three kinds of instruction generation: class declaration instruction generation, field fusions,
Algorithm 3 Single Node Generation

1: Map store=(IGraphNode obj, BytecodeQueue cn)
2: procedure visitNode(Object pivot, Class curr=pivot.getClass())
3: if !isEnableJIT(curr) then return null;
4: end if
5: if store.contains(pivot) then return store.get(pivot)
6: end if
7: Create an empty class node cn
8: emitClassNameInst(cn, curr)
9: for all field f in curr.fields do ▷ Field Fusion
10:   childObj = getChild(pivot, f)
11:   childClassNode = visitNode(childObj)
12:   fieldFusion(cn, f, childClassNode)
13: end for
14: for all Method m in curr.methods do ▷ method fusion
15:   for all instruction inst in m do
16:     if inst is a field instruction and isFusedInst(inst) then continue
17:     end if
18:     if inst is a potential method inlining inst then
19:       recMethodNode = getReceiverMethodNode(inst, pivot)
20:     end if
21:     if recMethodNode is null then
22:       cn.add(transform(inst)); continue;
23:     end if
24:     methodFusion(cn, inst, recMethodNode)
25:     continue
26: end if
27: cn.add(inst)
28: end for
29: postFusion(cn)
30: store.put(pivot, cn)
31: return cn
32: end procedure
and method fusions. For a class, GraphJIT scans its bytecodes sequentially, and fuses these bytecodes with the generated bytecodes from its children. The new fused bytecodes are appended to the \( cn \).

In this section, we take the object graph in Figure 6.5a as a running example to demonstrate how to generate bytecodes for node root’s \( cn \). We also show the decisions during bytecode generation. In the section, we use \( s \) and \( t \) to represent node 0 and 4 in Figure 6.5b, respectively, and assume that \( o4 \)’s \( cn \) has all generated bytecodes of \( o4 \).

### 6.5.2.1 Class Declaration Instruction Generation

The first bytecode instruction emitted for root’s \( cn \) in Algorithm 3 is a Class declaration. GraphJIT defines a new unique class \( newcls \), as a subclass of \( curr \), and adds this instruction to the \( cn \). The new class name \( newcls \) is generated by prefixing a string “DYN” and appending an integer number to existing class name in \( curr \).

### 6.5.2.2 Field Fusion

A field fusion is a substitution of a node by its children. Assume the node set \( V \) in a CCG is \( \{0,1,\ldots,v\} \), and the out edges for a node \( s \) is \( E_s = \{e_0, e_1, \ldots, e_{m(s)}\} \) and all children of node \( t \) are leaves. For two nodes \( s, t \in V \) and \( e_{s,t} \in E_s \), \( s(e_{s,t}) = t \) represents that \( s \) reaches \( t \) via the edge \( e_{s,t} \), and a set \( t(E_t) \) is \( t \)’s children \( (t(E_t) \subset V) \). \( E_s \) is updated by substituting \( e_{s,t} \) with
edges in the set

\[ F(e_{s,t}, E_t) = \{ e_{new} = \text{concat}(e_{s,t}, e'_t) | e'_t \in E_t \} \]

The function \( \text{concat} \) concatenates two-edge names by “\_”. After removing all edges that ends at \( t \), the node \( t \) is removed from \( V \). For example, both node 4 and the edge \( n1 \) connecting node 4 and node 0 in Figure 6.5b are removed, and two new edges \( n1\_n1, n1\_n2 \) are added to \( E_0 \) at the same time.

In the implementation of \( \text{fieldfusion} \), GraphJIT retrieves \( E_t \) from \( t \)’s visitor \( \text{child} \) directly. Then the \( s \)’s \( \text{visitor} \) adds new field instructions for each component in the set \( F(E_k, E_t) \). For example, to remove node 4 in Figure 6.5b, \( s \)’s \( \text{cn} \) adds two instructions shown in Listing 6.2.

```java
1 // access=protected+final, desc=LINode, and signature=value=null.
2 visitor.fields.add(new FieldNode(access, "n1\_n1", desc, signature, value));
3
4 visitor.fields.add(new FieldNode(access, "n1\_n2", desc, signature, value));
```

Listing 6.2: Field Fusion for node 4

to replace the original \( \text{new FieldNode(access, "n1", desc, signature, value)} \).

### 6.5.2.3 Method fusion

\( \text{Method fusion} \) refers to a merging of methods from two nodes, \( s \) and \( t \) \((t \in s(E_s))\), into \( s \)’s \( \text{cn} \). The method fusion is necessary since the node \( t \) is
removed from the class graph. In other words, the class s’s layout is modified.
The fusion can be in the form of either method inlining or method copying.
In the former case, a called method in t is inlined at a call site in s (no
method is newly created in cn), while in the later a called method in t is
cloned to s as a single method (a method is newly created in s’s cn). Along
with the method fusion, some instructions are transformed to adapt to the
node’s layout change.
In this section, we show how GraphJIT conducts method inlining and how
instructions are modified, while the decision for method selection will be
discussed in Section 6.5.2.4.

Method Inlining  Method inlining is an operation that replaces a method
call site with the called method’s body to reduce a method invocation’s
overhead. This inlining operation can be simply achieved by copying t’s
bytecodes at a proper location of s’s cn.
GraphJIT only targets a specific kind of method invocation instructions. A
bytecode method invocation instruction is

\[
\text{opcode owner name desc}
\]

where \text{opcode} is either \text{invokevirtual} or \text{invokeinterface}; \text{owner} is a class’s
internal name\(^4\) and both \text{name} and \text{desc} determines the declaration of the

\(^4\)The internal name of a class is its fully qualified name, as returned by
\text{Class.getName()}, where ‘.’ are replaced by ‘/’ [2].
target method. In GraphJIT, a method call site is recognized to be a potential candidate for inlining if its owner is the same as the base type of a node in a class graph.

The implementation of inlining operation is very straightforward (from line 20 to 24 in Algorithm 3). In the method `getReceiverMethodNode`, GraphJIT deduces the invocation receiver; checks whether the receiver’s class is excluded from inlining; and retrieves and returns the called method node of that class. Inside of method `methodFusion`, the instructions in `recMethodNode` are all appended to `cn` with some necessary modifications (e.g., variable renumbering).

```java
1 public <init>(LINode; LINode;)V
2 ...
3 ALOAD 0
4 ALOAD 1
5 PUTFIELD Node1.n1 : LINode;
6 ...
7 public invokeExact()Ljava/lang/Object;
8 ALOAD 0
9 GETFIELD Node1.n1 : LINode; //Get variable 0’s n1 field.
10 INVOKEINTERFACE INode.invokeExact()Ljava/lang/Object;
11 CHECKCAST java/lang/Boolean
12 INVOKEVIRTUAL java/lang/Boolean.booleanValue()Z
13 IFEQ L1
14 ...
```

Listing 6.3: bytecodes for Node1

---

5A method node is a sequence of bytecodes of a method body.
Instruction Modification in s  After field fusion, the node s's child t is replaced by t's children in the class graph. Corresponding to these changes, GraphJIT modifies instructions of methods in s, when these instructions are relevant to the fused field t, before they are added to s's cn.

These kinds of instructions are mainly field instructions (i.e., PUTFIELD and GETFIELD, GETSTATIC) and method invocation instructions. A specification of a field instruction in the JVM is

\[
\text{GETFIELD/PUTFIELD} \quad \text{owner.name : descriptor}
\]

which means to get/set a field value of an object reference, the internal name of which matches owner \[71\]. A GETFIELD instruction gets the field value of the object that is referenced by the top stack variable. A sample field instruction is shown on line 9 in Listing 6.3.

A field instruction in s is fusible if the object referenced by the target has been fused. For a GETFIELD instruction, the target is determined by both the variable name on the top of the operand stack and the field name that is defined in the current field operation instruction. Meanwhile, the owner in the field operation instruction should also be consistent with the variable type at the top of the operand stack. GraphJIT skips fusible GETFIELD instructions directly, because the corresponding field object have been fused during field fusion phase. Assume the bytecode shown in Listing 6.3 is a fragment of node 0 in Figure 6.5b and the internal name of the class represented by
$cn$ is $newcls$. The instruction on line 9 of Listing 6.3 is skipped, as the node 4 in Figure 6.5b does not exist any more after GraphJIT transformation.

Compared to fusible GETFIELD instructions, GraphJIT treats fusible PUTFIELD instructions differently. A fusible PUTFIELD instruction on $e_k$ is unfolded into a group of PUTFIELD instructions, which set each field in $F(e_k, E_t)$ with the corresponding field of the object, which is referenced by a variable at the top JVM stack, respectively. Meanwhile, the owner in these PUTFIELDs are all changed to $newcls$, which is the class name of bytecodes in $cn$. Thus, the constructor method in Listing 6.3 becomes that in Listing 6.4.

```java
1    public <init> (LINode; LINode;) V
2      ...
3    ALOAD 0  //Load this = newcls
4    ALOAD 1  //Load variable 1
5    ASTORE 2
6    ALOAD 2
7    GETFIELD Node1.n1  //Load variable 2’s n1
8    PUTFIELD newcls.n1_n1 : LINode;  //set 0’s n1_n1
9   
10   ALOAD 0
11   ALOAD 2
12   GETFIELD Node1.n2
13   PUTFIELD newcls.n1_n2 : LINode;  //set 0’s n1_n2
```

Listing 6.4: Bytecodes for $s’$ constructor after fusion
6.5.2.4 Node Fusion Criteria

To balance potential benefits and translation expenses, GraphJIT has to make a decision whether a graph node is suitable for a fusion or not. These benefits can be shorter paths from a graph root to leaves for the traversal. They can also be fewer JIT compilation tasks from MHs, and thus less competition for the JVM JIT compilation threads. Translation expenses occur at runtime, and these expenses include bytecode parsing, generation, and verification. These expenses would increase along with the increase of selected graph nodes for fusion. Besides, both MHG bytecode generation and JIT compilation are sensitive to the complexity of bytecodes, which can be related to the number of the selected nodes and the subgraph structure formed from these nodes. Therefore, nodes in a graph have to be selected for the fusion carefully, so that the performance gain can outweigh the cost for this gain.

Node Selection for Fusion  In GraphJIT, node selection criteria are based on field annotations (configurations), dynamic selection, and filters. Both field annotations (configurations) and filter APIs indicate what types and fields are eligible for fusion. This is especially useful when developers have prior knowledge which nodes are not suitable for fusion. Dynamic selection is based on a node’s runtime information.

- field annotations (configurations). In this approach, a class’s field is annotated explicitly to indicate whether the field is eligible for fusion. A
usage example of this annotation `AnnoBytecode` is shown in Listing 6.5 where the `name` defines the customized edge name from $s$ to $t$, rather than defined by the field name $n1$; the `Class` is the interface name or abstract base name; and `excluded` indicates whether the field is eligible for field fusion. The default value for `excluded` is false, meaning that GraphJIT would consider the node referenced by the annotated field for fusion. This annotation is retained at runtime, so that GraphJIT is capable of accessing it at runtime.

- Dynamic selection (i.e., counter-based selection and method-size-based selection). In this approach, a counter is associated with a graph node, and its value increases by one when the node is visited (e.g., one of its methods is called). A node is classified to be hot if the counter’s value is greater than `Threshold`. This approach is based on the assumption that the more a node is seen by GraphJIT, the more likely its methods are to be executed at runtime. The details of this part will be discussed in Section 6.5.3.9.

- Filter API. GraphJIT also provides an API to let developers specify what kinds of concrete classes should be excluded from bytecode gen-
eration. GraphJIT defines a global excluded class set. During translation, a CCG node is excluded from translation if this node (i.e., the class) is in the excluded class set. Compared to the field annotation solution, this approach provides a fast way to exclude a large number of CCG nodes from translation, which could be useful during performance tuning.

**Method Selection for Fusion**  Methods in $t$ are divided into three sets: $S_1$, $S_2$, and $S_3$. $S_1$ is a set of methods, which will be inlined to call sites; methods in $S_2$ are called by methods in $s$ or $S_1$ directly or indirectly, and these methods will be duplicated to $s$; and the remaining methods of $t$ are in $S_3$.

The union of $S_1$ and $S_2$, $S_1 \cup S_2$, is fixed for a pair of $(s,t)$, and all methods in $S_1 \cup S_2$ will be fused to $cn$ of $s$ by inlining or duplication. The criteria to separate $S_1$ and $S_2$ is based on a method’s size (i.e., the number of non-debug bytecode instructions in a method). During fusion, a node’s method will be in $S_2$, if the estimated fused method size would exceed a predefined threshold.

### 6.5.3 Design Overview

Based on the bytecode generation concept discussed in Section 6.5.1 this section provides GraphJIT’s design and implementation. GraphJIT’s design goals are threefold:
• Fast compilation. GraphJIT is a Just-In-Time bytecode compiler, and the compilation procedure should be as fast as possible so that the impacts (i.e., the pause time) on a JVM can be minimized. Thus, GraphJIT avoids expensive global optimizations, and turns off unnecessary operations (i.e., repeated bytecode validations and class loading), unless users explicitly require it.

• Adaptable and simple. The second design goal is to make GraphJIT adaptive and simple. GraphJIT is a translator for FTSDA graphs. It only provides a single public compilation API and a number of configuration options to control the compilation behaviors.

• High Concurrency. Concurrency is one of the significant features for modern programming languages. The concurrency supported by GraphJIT is threefold. First, GraphJIT does not violate a program’s concurrency and cause execution failure. Second, the bytecode generation in GraphJIT is thread safe, when GraphJIT is called in a multi-threaded environment. Third, GraphJIT itself supports concurrent bytecode generation at runtime. By the concurrent generation, a user thread is not required to wait until the compilation completes, so that the performance is maximized.

GraphJIT is implemented in the Java programming language, and its design rules are:

• Service Orient-Architecture (SOA). The design of GraphJIT follows
SOA rules [67], and all fundamental functions (e.g., class loading, cache, etc.) are encapsulated as services, and individual functions only focus on their own functions without interfering with other implementations.

- **Visitor pattern.** GraphJIT strongly depends on the bytecode manipulation library *OW2-ASM* [1], which is an all purpose Java bytecode manipulation and analysis framework. Based on the fact that the visitor pattern is the main design pattern in the ASM, GraphJIT also makes full usage of this pattern to make a seamless integration with ASM.

- **Plugin system.** GraphJIT also provides a number of interfaces to enable customized optimization behaviors. Any plugin can be integrated into GraphJIT, if the plugin implements the given abstract plugin, and can be activated only via configurations before GraphJIT starts. Via the plugin, a GraphJIT user can redefine method inlining and bytecode generation rules, as well as decision choices.

An overview of the GraphJIT system is shown in Figure 6.7. GraphJIT is mainly made up of two components: Plugin and Core. The plugin component is a collection of tools for supporting customized plugin implementations and their integration with GraphJIT. It also provides a default plugin and a method handle plugin that serves as a bridge between GraphJIT and the method handle template system.
There are three layers in the core component: the service layer, the middle layer and the visitor layer. The service layer is at the bottom, and it provides all basic services for other layers. The middle is a visitor layer, which is made up of a number of visitors to traverse bytecodes of classes and methods. A class visitor contains a bytecode queue $cn$ for transformed bytecodes of a Java class, which is discussed in Section 6.5.2 on that queue. The upper layer mainly consists of an engine component and two contexts: $MethodContext$ and $ClassContext$. The engine performs a FTSDA graph compilation and optimizations, while both $MethodContext$ and $ClassContext$ track temporary data for method fusions and class fusions, respectively.
6.5.3.1 Engine

The engine in GraphJIT accepts a single generation task and outputs a new object graph that is equivalent to the source graph. It also connects all visitors and services together. The overview of the engine is shown in Figure 6.8.

Generation Task The engine in GraphJIT accepts a data structure called a generation task task, which is made up of two fields: object and init. The
object is of IGraphNode type, and it is the source graph’s root. The engine in GraphJIT will traverse all nodes that are reachable from object for generation by default. The field init is a boolean value, instructing GraphJIT to construct a new instance for the generated bytecode class, if init is true. The init default value is true and this value is set to false when GraphJIT thinks the corresponding object should be fused to another graph node during traversal.

Engine Action Pipeline  In the engine, there is an action pipeline, which is made up of CheckCache action, Filter action, Timer action, Chooser action, Transformer action, ObjectSetter action, and UpdateCache action. Once the engine receives a task, it passes the task along this pipeline after pre-processing (e.g., creating ClassContext). The generation will be stopped if any action indicates that the object in the generation task is not suitable for compilation.

The CheckCache action mainly checks whether the given object or the graph with the same structure (i.e., a graph is represented by its root) in the task has already been compiled previously. The cached bytecode will be used directly, instead of moving to the next action, if the object is hit in the cache. Further a new instance is newly initialized if the corresponding init is true.

Follow CheckCache action, both Filter and Timer actions determine whether the object is suitable for the generation. The Filter action checks configured
information, to see whether the type of the given `object` is excluded from compilation. Similarly, `Timer` action compares the `object`’s Invocation Counter (IC) to a threshold, and the `object` will not be compiled if its IC is less than the threshold.

The existence of `Filter` and `Timer` actions is intended to trade off the expense of a graph compilation against the potential benefits obtained from the compilation. First, static configurations (e.g., JVM options, command lines or external configuration files) that exclude some kinds of graph nodes from compilation in `filter` provide a means to tune the performance statically, and this is especially useful when users have prior knowledge about the potential benefit of special node compilation. Second, the dynamic node selection for compilation is based on the assumption that an object will be still executed in the future, if it had been frequently called in the past. Intuitively, it is worthless to compile an object that is rarely executed.

After `Timer` action, the `Transformer` action conducts an actual compilation. It creates a new empty bytecode queue `cn` for the `object`, and then a number of class visitors and method visitors, which iterate over components of the `object`’s class, as well as its children in the CCG.

Both `ObjectSetter` and `UpdateCache` actions are next to the `Transformer` action. In the `ObjectSetter` action, nothing will be done if the `init` field in the task is false (GraphJIT does not create any instance of the newly generated bytecode in `cn`). Otherwise, the bytecodes in the `cn` are loaded first. Then an instance of the loaded class is created, and its field members are set to
the children in the object of the generation task. Finally, bytecodes in the cn, wrapped as BytecodeResource, are inserted into the cache as a (object, BytecodeResource) pair.

6.5.3.2 Engine Transformation

Token passing In the Transformer action, GraphJIT creates a number of visitors to do a graph fusion. The bytecode queue cn is shared among these created visitors, but manipulated by them sequentially.

As the main operation on a cn is appending, GraphJIT maintains a token for each bytecode queue cn. A visitor is capable of appending bytecodes to the cn only if it owns the corresponding token. GraphJIT is responsible for managing the token, and passes it from a method visitor (Vs) to another method visitor (Vt), when GraphJIT wants to fuse two methods (e.g., method inlining or method copy) from different nodes.

Receiver Type Determination To complete inlining, GraphJIT needs to determine the receiver type of a method invocation. Due to language polymorphism, the receiver type is not determined during the compilation phase. Take the invocation instruction

\[
\text{INVOKEINTERFACE INode.invokeExact()}\text{Ljava/lang/Object}
\]

in Listing 6.3 for example; the receiver type can be either Node1 or Node2, both of which implement the interface INode.
To determine the actual receiver type, GraphJIT takes two rounds for a given method. In the first round, GraphJIT analyzes bytecodes and infers the receiver types for individual method invocations. With the first round’s result, GraphJIT scans bytecodes and conducts inlining optimizations for invocation instructions in the second round.

GraphJIT in the first round mainly builds a method visitor `typeInferencer`, which has a private method visitor. After first round, the `typeInferencer` outputs a frame dependence map, `typeIndex`, which tracks a method invocation frame (i.e, a bytecode instruction) with the actual receiver type of that invocation instruction. This dependence map is saved in the `MethodContext` and will be looked up, when GraphJIT enters the second round for inlining optimization.

There are two steps to construct a `typeIndex`:

- Analyzing frame dependence. For a given bytecode method, GraphJIT in the first round builds a dependence tree, a structure similar to an AST, by analyzing bytecode sequence syntax. This dependence tree can be mapped from an AST tree by a) replacing a node operation with a bytecode frame id and b) removing a node from the tree, if none of its parents (direct or indirect) is an invocation instruction.

For example, the bytecode Listing 6.6 that launches an invocation on an array’s member would generate a dependence tree in Figure 6.9, where the node id is the frame index in the sample code.
Dependence Tree’s Evaluation. For a dependence tree, GraphJIT evaluates it in post-order traversal. Since, all leaf nodes in a dependence tree are frames to load variables (e.g., 0, 1, 2,), the actual receiver, as well as its type, for a dependence tree’s root can be uniquely determined when the evaluation completes.

For a method invocation, GraphJIT saves the frame id (i.e., pc) and the computed receiver type as a (key, value) pair in the current Method-Context. This saved information will be used in the second round for inlining optimization.
6.5.3.3 Engine Optimizations

Two major optimizations that GraphJIT performs are graph inlining, discussed in Section 6.5.1, and inline caching.

Inline caching is a prevalent optimization in compiler systems. Its main idea is to remember receiver types that were resolved at a call site previously. Normally, methods in these remembered types would be intensively optimized and would be reused directly, if a real target in the future matches a remembered type. The benefit of this optimization is that the runtime can directly use the optimized method to avoid method resolution cost.

The inline caching optimization in GraphJIT addresses a mutable graph node, the children of which can be modified during runtime. For a mutable node, GraphJIT remembers the current optimized receiver, when the compilation happens, and encodes this receiver at the bytecode level directly. For a volatile child node, GraphJIT creates a new field member, *cached, referring to the remembered node, and then uses the cached one as a test. For example, a dynamic MH call site callsite.getTarget().invokeExact() has one method invocation invokeExact. The decompiled sources for the produced bytecodes, when inline caching is enabled, are shown in Listing 6.7.

6.5.3.4 Cache Hierarchy

As discussed in Section 4.4, a number of equivalent method handle graphs are created by the JRuby interpreter. In order to avoid regenerating bytecode methods for equivalent MHGs, GraphJIT provides a cache to store all
1 public class DYNGuardWithTestHandle821
2 extends BaseTemplate {
3     public static final CallSite guard_site = ..;
4     public static final MethodHandle guard_cached = ..;
5     ...
6     public final IRubyObject inlinedMethod(ThreadContext threadContext, IRubyObject iRubyObject, IRubyObject iRubyObject2, IRubyObject iRubyObject3) throws Throwable {
7         MethodHandle methodHandle = guard_site.getTarget();
8         if (methodHandle == guard_cached) {
9             // inline guard_cached.invokeExact(...) here
10             ...
11             return someObj;
12         } else {
13             return methodHandle.invokeExact(...);
14         }
15     }
16     ...
17 }

Listing 6.7: Illustration for decompiled sources with inlinecache

generated bytecodes of a graph node, and reuses them if necessary in the future.

The CacheService in GraphJIT is a service to manipulate the generated bytecodes. In the CacheService, a generated bytecode is organized as a (key, value) pair, in which the key is a graph node, and the value is a data structure BytecodeResource. The key uniquely identifies a subgraph structure after overriding graph node’s equal and hashCode methods, and it is generated from the root of that subgraph. The definition of the BytecodeResource class is shown in Listing 6.8. In this definition, the core field is _optimizedClassNode, which refers to a bytecode queue cn, and the _newObject is an instance of the class that is represented by the _optimizedClassNode.
CacheService provides an interface ICache for customized cache repository implementations. Based on this interface, developers are capable of designing their own cache repositories (e.g., both key and value structure) and purge strategies. As a default, GraphJIT itself only provides a global cache repository, where a global concurrent hash map is used to host all (key, value) pairs. The key, which is a graph key, is extracted from the root of a graph, while the value is still a BytecodeResource.

For the put operation, CacheService passes two arguments, i.e., a subgraph root object and the corresponding BytecodeResource, to the cache repository. The cache repository then transforms both arguments to the formats that the repository requested. For example, the default global cache extracts
the graph key from a graph root object, and then inserts it together with

\textit{BytecodeResource} to the concurrent hash map in the global cache. It is the
same for the \textit{get} operation.

In case the occupied cache size becomes too large, the \textit{CacheService} also pe-
riodically purges the cache repository. The purge policy is that the bytecode
resource with minimum hit counter value is first purged. Here, the counter
\texttt{counter} is increased by one, when the corresponding resource is cache-hit.

For MHGs, the \texttt{counter} value of all resources is reset to 0 at the end of each
purge period for the following reasons:

\begin{itemize}
  \item MHG locality. An MHG is created and executed, and the same MHG
  with the same method type will be created and executed soon. This is
  because of the locality of dynamic method invocations.
  
  \item MHG trend. During our profiling, MHGs are always created for dy-
  namic method invocations at the early stage. Later, most of MHGs,
  produced by GraphJIT, are JITted by the native JIT compiler, and
  further inlined to the dynamic call sites directly. In other words, the
  interpreter is no longer creating any new MHG in the latter execution
  stage, and zeroing the \texttt{counter} causes these cached bytecodes to be
  recycled more easily.
\end{itemize}
6.5.3.5 Class Loading and Bytecode Verification

**Class Loader**  *ClassLoaderService* in GraphJIT provides two methods to load generated bytecodes so that they can be recognized by JVMs.

The first method is a bytecode class loader, *BytecodeClassLoader*, which is of a standard Java class loader. The *BytecodeClassLoader* accepts a byte array and a string name, *classname*, as parameters, and initializes the content of this byte array to a Java Class. As a standard class loader, the *BytecodeClassLoader* adds *classname* to its constant pool, when the class loading for the given byte array succeeds.

Since a standard Java class loader does not support a class with multiple versions, the class loader system in GraphJIT adopts a *BytecodeClassLoader*-per-version policy, when the first method is chosen. In this policy, a *BytecodeClassLoader* remembers all *classnames* that it has loaded. For a class loading job, the class loading system chooses a non-conflict *BytecodeClassLoader*, which has not seen the requested *classname* in the job so far, to perform the task. The system has to create a new *BytecodeClassLoader*, if all existing loaders conflict.

The other method for a class loading task is *ACLoader*, which strongly depends on the method

\[
\text{sun.Misc.Unsafe.defineAnonymousClass(Class Hoster,}
\text{byte[]}\text{ bytecodes, Object[]}\text{ cpPatch}).
\]

---

6This refers to a class, the bytecodes of which can be modified at runtime.
The AClassLoader statically specifies the Hoster argument as AClassLoader.class, and argument \( cpPatch \) is left null, which indicates no modification on the constant pool. All dynamically loaded classes by AClassLoader are anonymous classes within the Class AClassLoader, so that multiple versions of a class can be repeatedly loaded by a single instance.

GraphJIT configures AClassLoader as the default, and the generated bytecodes are loaded only when the GraphJIT has to create an instance of the class represented by the bytecode queue \( cn \). The choice of AClassLoader is motivated by

- avoiding creating multiple class loaders. Different versions of a class can be loaded repeatedly via AClassLoader, as all loaded classes are anonymous classes, and the corresponding class names would not show up in the class loader hashtable.

- getting rid of security checks. According to the JVM specification, no user class is allowed to be in the protected \( java.lang \) package. However, the generated classes for method handles in our evaluations are required to be in these standard packages, so that they can be fully trusted by TR JIT compiler in J9 and be integrated directly with IBM’s specific JSR292 implementation.

**Bytecode Verification** Although both Java class loaders, AClassLoader and BytecodeClassLoader, have an embedded functionality to verify bytecodes prior to class loading, GraphJIT provides its own bytecode verifier that de-
pends on ASM library. The reason for an independent bytecode verifier is that GraphJIT needs a flexible and independent verifier that can be used at any time, instead of at the final class loading phase. GraphJIT uses the verifier to check the correctness of intermediate bytecodes, so that errors can be detected early before they are fused and optimized.

The verification is based on CheckClassAdapter from ASM and StackMapTable of a method in a class. For a method invocation instruction, GraphJIT compares its signature and deduced object types that are at the top of the operand stack. GraphJIT also checks variable types in the stack at the positions that are targets of jumps, and compares them to the types defined in the StackMapTable. If the verification fails, GraphJIT will discard the current bytecodes, and raise a GraphJIT exception for users.

The verification is only one pass, and it only targets methods in a generated class. In order to reduce the performance overhead at runtime, the verification is disabled by default, and users can turn it on for debugging, if the class loading of fused bytecode fails.

### 6.5.3.6 Plugin System and Interface

The plugin system in GraphJIT allows users to control two kinds of mappings:

- mapping from a graph node to a class node (or method node). For a graph node, the plugin system determines how to obtain its bytecode (i.e., a class node) at runtime. The default plugin implementation is to read the bytecode stream from a class’s class loader.
mapping from a method invocation call site to a method node. When GraphJIT sees a method invocation instruction and is about to perform inlining, it needs to get the body of the called method, in the form of a method node. The default plugin looks up the cache for the target receiver. If it is a cache hit, GraphJIT picks up the cache bytecodes (i.e., a class node), and extracts the called method node from it, according to the given method name and its descriptor. Otherwise, it requests GraphJIT to compile the receiver of this call site, as the last attempt for the bytecodes of the called method.

A minor difference between the two mappings is that the method node in the latter has already been inlined (i.e., fusion of a child graph), while the method node of a class node in the former only represents a single graph node. As shown in Figure 6.10, the mapping from the graph node $o_4$ to the class node $cn_4$ is the first kind of mapping, while the mapping from a
subgraph $G_{o5}$ to the method node $mn.5$ is the second kind of mapping.

GraphJIT provides two plugins: `SimplePlugin` and `MethodHandlePlugin`, both of which implements the plugin interface in Appendix A.2. The former is for general purpose mapping as default, and the latter adapts GraphJIT for the method handle graphs. To customize mappings, users have to declare their own plugin as a subclass of `AbsPlugin` and override abstract methods. Users also have to activate it by declaring the plugin type during the GraphJIT initialization phase.

### 6.5.3.7 Synchronous and Asynchronous Generation

GraphJIT works in two modes: synchronous and asynchronous generation.

**Synchronous Generation** In synchronous generation, a user thread has to wait for the completion of compilation when it calls the GraphJIT generation API. In other words, the generation might increase the execution time if the compilation time expense outweighs the speedup of the executing
In this mode, GraphJIT works in the user’s thread. Each request for a graph generation in GraphJIT is equivalent to creating a new engine instance. In a multi-threaded environment, an engine is lock-free itself and it can be fully shared among multiple threads, to maximize GraphJIT’s throughput.

**Asynchronous Generation**  In order to reduce the potential overhead of creating and recycling engines, and the blocking time that a user thread has to wait for graph compilation, GraphJIT can compile graphs asynchronously, in a way that both engine threads and user threads are concurrent.

The main component when GraphJIT is in this mode is a compilation task queue (i.e., generation task queue), as shown in Figure 6.11. In this mode, a compilation task is created and appended to the queue immediately when a user thread requests to compile a graph and the root of the graph is missed. GraphJIT has an engine thread pool, which manages the lives of engine threads for compilation. Each engine thread itself holds an engine reference and polls a compilation task from the queue, if the queue is not empty.

An engine thread has two states: running and blocked. An engine thread is in a running state, when it has a compilation task from the queue and is working on compilation, as discussed in Section 6.5.3.1. The thread state becomes blocked after the engine finishes compilation, but there is no task on the queue.

The main advantage of GraphJIT in this mode is its performance, especially
when a generation task would take a long time. When GraphJIT switches to asynchronous generation, a user thread returns directly without waiting for the completion of the compilation. In the meantime, the generation task is conducted in parallel with user threads and the generated bytecodes (i.e., a class node) are pushed to the cache when generation completes. Later, GraphJIT will initialize a new graph from cached bytecodes, and returns it back to the caller, if the graph structure associated with the graph node in the generation task has been seen previously.

6.5.3.8 Concurrency

GraphJIT’s concurrency lies in three facts:

- It supports asynchronous bytecode generation, which has been covered in Section 6.5.3.7.

- GraphJIT is thread-safe. GraphJIT encapsulates two main tree structures from ASM: ClassNode and MethodNode, as AtomicClassNode and AtomicMethodNode, so that both are adapted to multi-thread applications. When GraphJIT works with bytecode generation for method handle graphs, there is only one GraphJIT generator instance, and it is thread safe, regardless of multiple user threads in tests.

An AtomicMethodNode is a bytecode queue discussed in Section 6.5.2, and the access to its bytecodes is synchronized when it is shared by 

---

If not declared explicitly, both class nodes and method nodes in the whole dissertation refer to atomic class node and to atomic method node, respectively.
multiple visitors. The reason for the synchronization is that every method node has a label node list, and the access of this label list is not thread-safe for visitors when they see label related instructions (e.g., visitlabel and jump).

- Cache Concurrency. During asynchronous Generation, the cache would be frequently used, when multiple worker threads perform fusions concurrently. To reduce synchronization granularity, the concurrency support in cache is completed in the real implementation of the interface ICache, instead of CacheService. The default cache implementation, i.e., global cache, uses ConcurrentHashMap to host the key and BytecodeResource.

6.5.3.9 Graph Node Selection Policies

Both Timer action and Chooser action determine whether a node obj is suitable for GraphJIT’s compilation, by checking its hotness (i.e., an MH’s invocation count) and method size. If either hotness is less than threshold or the method size exceeds another threshold, the compilation on the obj would not be launched. GraphJIT allows customized selection policies implementations via plugins. The existence of these two actions is to exclude some unnecessary compilation effort at runtime.
Hotness threshold determination  The threshold is less than TRJIT’s warm JIT compilation’s threshold. In our model, it is chosen in a way that the majority \((p)\) of MHs’ invocation counts are above the threshold, when first warm MH JIT compilation. \(p\) is also configurable and its default value is 0.95.

Maximal method size  Based on the hotness, a node is likely to reach threshold, if it has more than one parent, and the corresponding source bytecodes might be inlined multiple times in the same queue bytecodes. This would result in a large class, and some JIT compilers (e.g., TRJIT in J9) are sensitive to the method size. Thus, GraphJIT lets users configure the maximal method size to avoid a large method body.

6.6 Method Handle Template System

The template system provides bytecodes for individual method handle graph nodes. A template is a code fragment. For a method handle, the execution of its template will emit bytecodes (i.e., a class node), the execution of which is equivalent to the execution of the MH itself. For an MHG, GraphJIT fuses bytecodes that are generated by MHs’ templates into a single or multiple new MH transformations.

\[^{8}\text{TR JIT refers to a JIT compiler that translates bytecodes into machine codes in IBM J9 JVM.}\]
6.6.1 Generated Class Layout in Templates

The implementation of method handle templates is specific to IBM J9. In the `java.lang.invoke` package, each MH transformation is represented by an MH class, which extends the abstract class `MethodHandle`.

A generated class (i.e., generated bytecodes and new MH transformation) from an MH’s template is still of a method handle type. To distinguish from existing transformations, the name of the newly generated class is concatenated with three components: prefix, original MH’s class name, and an atomic index, where prefix is a configurable string (e.g., “DYN”), and the atomic index is used to guarantee the transformation’s uniqueness.

The generated class’s layout is shown in Listing 6.9. The generated MH transformation class consists of: a class declaration, fields, two methods, and a static initializer `<clinit>`. The super class of the generated MH class is an abstract `BaseTemplate`, which extends `PassThroughHandle` and encapsulates specific J9 JIT operations. `PassThroughHandle` is a sub-class of `MethodHandle`, and has a `MethodHandle` field, named `equivalent`. During interpretation, a `PassThroughHandle` MH passes the invocation to its `equivalent` children.

Besides generated fields, the generated MH transformation class contains a transformation method and a thunk method. The transformation method, named `inlinedMethod` by default, is an ordinary Java method that is generated based on the source MH’s transformation, while the thunk method is specific to TR JIT in J9, and it adapts the transformation method to the JIT compiler in J9. The function `<clinit>` is responsible for initializing the
static fields in the generated class. GraphJIT concentrates on the generated class’s fields, the transformation method, and its <clinit> method.

```java
public class className extends BaseTemplate{
  //new static and non-static field members.

  //Transformation method. Its modifier is determined by configurations.
  modifier inlinedMethod(args...){
  }

  //thunk method
  public final java.lang.Object invokeExact_thunkArchetype_L(
    args, int) throws java.lang.Throwable{
    ....
    invokevirtual/invokespecial inlinedMethod(args)
  }

  <clinit>{
    //init static field members
  }
}
```

Listing 6.9: Generated Bytecode Layout
6.6.2 The Role of Method Handle Template

Method handle templates are a set of Java methods to construct and provide bytecodes for individual MHG’s nodes. For a method handle, the corresponding template’s execution will emit bytecodes representing the MH’s transformation. The emitted bytecodes are in the form of a class node, which is encapsulated as a Blob structure (the definition is in Appendix A.3), and is returned back to GraphJIT.

The relationship between method handles and a template is shown in Figure 6.12. A template is only associated with a method handle transformation (i.e., a method handle class in J9), and instances that have the same transformation share the same template. During a template’s execution, bytecodes are emitted according to the method types and other optional data of individual MH instances. In other words, outputs of templates’ execution are different if a) these templates are associated with different MH transformations, or b) templates are the same (MHs have the same transformation), but method types or optional data of these MH instances are different. As shown
in the figure, there are four MH instances of the same transformation on the left, and two MHs with the red color have the same method type. After template execution, three class nodes are generated on the right, in which the class node for the type1 represents the transformation of both MH1 and MH2.

6.6.3 Template Overview

The template system is implemented inside of the Java standard package `java.lang.invoke`. In this package, the abstract class `MethodHandle` implements GraphJIT’s interface `IGraphNode` in Listing 6.10, via which the method handle plugin combines both GraphJIT and the template system together.

```java
1 interface IGraphNode{
2     public Blob get(String clsName, String mhName,
3                      String desc, boolean initInstance).
4     ...
5 }
```

Listing 6.10: Interface IGraphNode

The API `get` is the entry for the template system, and it returns bytecodes representing the transformation after execution.

Similar to GraphJIT, the main component in the template system is an action chain, as shown in Figure 6.13. Once the API `IGraphNode::get()` is called, the cache checker first checks whether the template with the same method type, as well as optional data, has been executed before. If not, then
both field generator and method generator are called to emit instructions of a class’s fields and methods. After the method generator, both verification and class loader verify and load the emitted bytecodes to ensure the bytecodes’ correctness. The final action is cache updater that puts the newly generated bytecodes, as a Blob, into the local cache. All steps, except Field generator and method generator, can be disabled by configuration for generation efficiency.

6.6.4 Template Implementation and Management

Field Generation The field generation aims to emit bytecode instructions to create class fields. The creation of these fields is based on the structure of the source method handle node in J9.

The decision of what kinds of fields should be created is determined by the source MH transformation and user’s configurations. All fields of the MethodHandle type are included if these fields are required by the transformation. Thus, the template system reuses the annotation AnnoBytecode in Listing 6.11 provided by GraphJIT, to label fields of source MHs. For an
existing MH class definition in J9, the template system reads an MH class
definition at runtime and analyzes its field members. For a field, the tem-
plate system emits a field creation instruction, for the new class node, if the
field is annotated with AnnoBytecode and its excluded value is false. The an-
notation is copied to the generated fields, so that GraphJIT can understand
its meaning during compilation. For example, a created field with true value
of the excluded directs GraphJIT not to inline the corresponding field.

```java
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target(ElementType.FIELD)
3 public @interface AnnoBytecode {
4     public String name() default "next";
5     public Class type() default MethodHandle.class;
6     public boolean excluded() default false;
7 }
```

Listing 6.11: AnnoBytecode Annotation

For all MH classes in J9, the AnnoBytecode is used to label all method handle
fields. In Listing 6.11, name() refers to a field’s name, and the type() is the
field’s class type, which is MethodHandle.class by default. During the field
generation phase, a field creation instruction is added to the corresponding
class node cn, if AnnoBytecode is found and its excluded value is false.

**Method Generation**  The transformation method contains bytecodes, the
execution of which is equivalent to the execution of the source MH. For ex-
ample, method handle class FilterReturnHandle contains two MethodHandle
fields: \textit{filter} and \textit{next}, and the transformation method for a \textit{FilterReturn} transformation type should be equivalent to the invocation

\[
\text{return } \text{filter.invokeExact}(\text{next.invokeExact}(\text{arguments}));
\]

The body (i.e., generated bytecodes) of the transformation method is a combination of executions on source MH’s fields, and it is determined by the source’s transformation and its method type.

A transformation method’s constructor is made up of method declaration and instruction generation for the method body. A method declaration consists of a modifier, a method name, and a method descriptor. Here both method name and modifier are determined by configurations at template system initialization, and the descriptor is converted from the MH’s method type.

For instruction generation for a method body, an MH’s method type determines the number of variable loading instructions, instruction selection, and invocation description for \textit{invokevirtual} (or \textit{invokeinterface}). For example, to generate an invocation on a method type \textit{(int, String)Object}, two argument variables, which are \textit{int} and \textit{String} type, have to be loaded onto the stack first. This requires the emission of two instructions (\textit{iload} and \textit{aload} respectively), as shown in Listing\ref{lst:method-instructions}.

\begin{verbatim}
1 //load receiver first.
2 iload 1
3 aload 2
\end{verbatim}
The other generated method is the thunk method, which is specific to the J9 method handle JIT implementation. For a method handle, the TR in J9 retrieves its thunk method and compiles it to machine code. Thus, the way to generate a thunk method (i.e., method name, method description, and method body) has a set of patterns. A thunk method’s body mainly consists of instructions that trigger JIT compilation and transfers the invocation to the corresponding transformation method. The connection between the thunk method and the transformation method of a generated class is the last method invocation instruction, as shown in Listing 6.9. To complete this method invocation, the thunk method’s arguments are cast according to the transformation method’s descriptor, and loaded into the stack first. Then the template system selects an invocation instruction (e.g., `invokevirtual`, `invokespecial`, and `invokestatic`) for call transferring.

**Template Cache** The template system maintains a local cache to manage the emitted bytecodes (i.e., class nodes). This local cache, called `ThunkBlobs`, is only for internal usage to avoid generating the same class node repeatedly. Similarly, `ThunkBlobs` organizes all generated bytecodes (i.e., class nodes) as (`TemplateKey`, value) pairs, where the value is a `Blob` that encapsulates the generated class node. Here, a `TemplateKey` is a method handle key that is a combination of the transformation type, method type, and other optional...
Inside of the template system, the cache *ThunkBlobs* is looked up to check whether the target bytecode has been generated some time previously. At the end, the newly generated bytecode resource is put into the local cache with its *Templatekey*.

To avoid potential contention on a single *ThunkBlobs*, the local cache is divided into multiple segments according to transformation types. Each transformation type maintains its distinct *ThunkBlobs*, so that the contention on *ThunkBlobs* of different transformations is directly eliminated. Additionally, the template system provides an option to let users turn off the local cache directly, so that the generation can start from scratch. The justification for disabling local cache is that the generated bytecodes for a single method handle node might be trivial when compared to the contention on the local cache. This is because the size of an MH’s transformation method within MH is always small.

### 6.7 Template System, GraphJIT and JSR292 Integration

The integration between the template system, GraphJIT, and JSR292 is shown in Figure 6.14. In the figure, there are two main spaces: JSR292 space (i.e., package *java.lang.invoke*) and GraphJIT space.
6.7.1 GraphJIT

In the GraphJIT space, GeneratorFactory manipulates all created generators (e.g., the default generator, the method handle generator, etc.) at runtime. For each kind of generator, there is only one unique version in the factory. The method handle plugin in GraphJIT space serves as a bridge, connecting both GraphJIT and individual method handle nodes. It provides GraphJIT with bytecodes associated with a single method handle node, and is injected into the method handle generator when the generator is first created. The method handle plugin was originally implemented in the JSR 292 package, but was migrated to GraphJIT package, as a built-in plugin, for the convenience of testing. Another important explanation for this migration is
that the method handle plugin only depends on GraphJIT’s common interface, instead of any specific JSR 292 APIs. For example, the type that the method handle plugin can understand is \( I\text{GraphNode} \), while an MHG node is a method handle, which is of \( I\text{GraphNode} \) type.

The JSR 292 space is atop the GraphJIT space. Inside the JSR292 space, a method handle GraphJIT generator is first retrieved from \( \text{GeneratorFactory} \), and this reference is kept locally and shared among all \( \text{invokedynamic} \) call sites. Later, all MHGs created by bootstrap methods are compiled by this cached local generator and the compiled MHG versions are linked to call sites or set as other MH targets.

### 6.7.2 JIT Compilation of the Generated Bytecodes

With GraphJIT and the template system, the whole work flow for an MHG compilation is shown in Figure 6.15. First, each node is translated into a class node via the template system. Then the transformation methods in

---

9In this section, we use a \( \text{GuardWithTestHandle} \) MHG for example and assume all nodes, except MH terminal nodes, in an object graph \( G_{\text{root}} \) are available for compilation.
Figure 6.16: Illustration: Set equivalent to be $G_{root}$
Figure 6.17: Illustration: Set *equivalent* to be the MH Referencing the Transformation Method
these class nodes (i.e., $C_{\text{root}}$, $C_4$, and $C_5$) are fused together as a single large class, named $\text{DYNGuardWithTestHandle0}$, by the GraphJIT compilation. In other words, the transformation method in the final $\text{DYNGuardWithTestHandle0}$ is a fusion of all the graph’s transformation methods that are generated by the template system. Meanwhile, GraphJIT also initializes an instance, $mh_{\text{new}}$, of the generated class $\text{DYNGuardWithTestHandle0}$, and sets its children (i.e., $o1$, $o2$ and $o3$). Based on the configuration, GraphJIT also sets $mh_{\text{new}}$’s equivalent to be either a) the root of source graph $G_{\text{root}}$ (as shown in Figure 6.16) or b) a new method handle that binds to transformation method (as shown in Figure 6.17). In Figure 6.16, the root of the generated MHG is $g1328293199$ has five children, four of which reference leaves via edges $\text{trueTarget\_guard\_next\_next}$, $\text{trueTarget\_trueTarget\_next\_next}$, $\text{trueTarget\_falseTarget}$, and $\text{trueTarget\_falseTarget}$. The source MHG for this graph is referenced by the edge $\text{equivalent}$.

The template system provides three means to enhance a generated MH for JIT. First, an invocation count of a generated MH instance is modified to be a large value for aggressive JIT optimization. Second, an option for assignment of the $\text{equivalent}$ field on the generated MH class is provided. This is because all generated classes are of $\text{PassThroughHandle}$ class, and these instances would transfer invocations to their $\text{equivalent}$ children, when they are in interpretation mode. Therefore, users can configure the $\text{equivalent}$ to be either a) the source MHG $G_{\text{root}}$ or b) a new direct MH that binds to the transformation method of a generated MH class, so that the generated MHG
can be fully warmed up even in interpretation phase.

Although newly generated MHs are new transformations and does not exist in JSR292, the execution of these MHs still conforms to the IBM implementation and JSR292 specification. The explanations are that:

- the only way to execute a method handle is to call its invocation method (i.e., `invokeExact`, `invoke` and `invokeWithArguments`) according to the JSR 292. The generated MHs inherit this method from their super class `MethodHandle` and it is illegal to call other methods from external users.

- it is compatible with method handle JIT compilation in J9. For a method handle, the TR only treats its thunk method as a compilation candidate. It starts compiling the thunk method when the number of executions of this method handle exceeds a threshold.

### 6.8 Object Cache and Class Cache

For performance reasons, two alternative caches: the *object cache* and the *class cache*, are provided in JSR292 space\(^\text{10}\) beside GraphJIT’s default global cache repository. Both caches are different from the local cache in the template system described in Section 6.6.4 in that both are for fused graph nodes that are generated by GraphJIT. Both caches are implemented in JSR

\(^{10}\)A JSR292 space is the Java standard package `java.lang.invoke`, which is dedicated to JSR 292 implementations.
space and used by GraphJIT.

For the object cache, generated bytecodes (i.e., a class node or BytecodeResource) are remembered by a volatile field of the source MH, the graph node that the bytecode resource is generated from. Since the cached BytecodeResource is only owned by a single MH instance, the object cache is lock free. The class cache is a little different from the object cache. A class cache is maintained by a transformation type (i.e., a final sub-class of the MethodHandle), and shared by all MHs that have the same transformation type. In a class cache, a source MH and a BytecodeResource are formed as a (key, value) pair. Here the definitions of both key and value are the same as that in the global repository in Section 6.5.3.4. Compared to GraphJIT’s global repository, all generated class nodes in a class cache have the same transformation type that is consistent with the class, where the class cache is. The concept of class classes is the partition of global repository in CacheService into multiple small regions, so that the access contention on a single global repository can be mitigated.

The choice of cache modes for GraphJIT can be achieved by configuration. The global repository is used as the default. The full configurations is listed in Appendix A.4.

6.9 Evaluation

The evaluation answers the following research questions:
• By how much can memory use benefit from the MHG compilation technique?

• By how much can execution time be improved for individual tests?

• By how much can JIT compilation cost be reduced?

• What is the expense for the MHG compilation technique at runtime?

• What is the correlation between performance speedup and other factors (i.e., TR JIT compilations, MHGs compiled by GraphJIT, and filters)?

In this section, the benchmark for the evaluation is still CLBG JRuby-micro, and the configurations for GraphJIT are left as default, the values of which are in Appendix A.4 unless they are explicitly declared.

6.9.1 Memory Use Impact

Figure 6.18 shows the percentage of GC pause time reduction. The figure shows that the GC pause time increases (i.e., 10.17% on average) when GraphJIT is enabled for MHG compilation.

The explanation for this result can be from two perspectives. First, more objects are available for GC, when GraphJIT is enabled. These objects are temporary objects created by GraphJIT during translation, and further objects are internal MH nodes fused by GraphJIT. The fused internal MH nodes are obsolete once they are substituted by MHs produced by GraphJIT for execution. Second, the difference in GC pause time reduction on different
Figure 6.18: GC Pause Time Comparison

(a) GC Pause Time

(b) GC Pause Time Reduction (Mean: 0.9%)
tests is caused by MHG patterns (i.e., the distribution of different MH transformation combinations). The generation for some kinds of transformations involves more temporary objects than others. For example, there are more frequent combinations (e.g., GuardWithTestHandle and ReceiverBoundHandle) in cal and so_list than that in the other tests, and the large existence of these transformation combinations does not favor GC pause-time reduction and execution time speedup.

6.9.2 Execution Time

A test’s execution time, ExecTime, is measured as the elapsed time to complete the test. We use ExecTime$_{orig}$ and ExecTime$_{GraphJIT}$ to represent the execution time without and with GraphJIT for a Ruby test. Thus, compared to ExecTime$_{orig}$, ExecTime$_{GraphJIT}$ includes

- a newly added expense on GraphJIT compilation;
- expenses that the JVM interprets the generated bytecode from GraphJIT;
- expenses that the JVM executes the jitted code of the generated bytecodes from GraphJIT; and
- an increased GC pause time.

The TR JIT compilation expense is not counted, since TR JIT compilation is done asynchronously.
Figure 6.19: ExecTime Overview
Another measure, *Speedup*, is also defined as

\[
\frac{\text{ExecTime}_{\text{orig}} - \text{ExecTime}_{\text{GraphJIT}}}{\text{ExecTime}_{\text{orig}}} \times 100\% \quad (6.1)
\]

### 6.9.2.1 Micro-indy CLBG

Both *ExecTime* and *Speedup* for the JRuby micro-indy CLBG are shown in Figure 6.19. According to the figures, the mean speedup for all benchmark tests is 2.9%. For the JRuby micro-indy benchmark, GraphJIT only contributes a minor speedup.

However, GraphJIT’s impact on a test’s execution varies. Table 6.1 groups tests by their speedups. In the table, 42% of tests have positive speedups with maximal 67% speedup, while 39% of tests have negative speedups. The distinction between tests with positive and negative speedups is caused by various factors (e.g., the distribution of transformation types, TR JIT compilations). For example, in some cases, a new MHG is generated by GraphJIT, but never used, nor JITted by TR, at runtime, while compilation effort spent by GraphJIT exceeds the reduced interpretation cost on the source and target MHGs. In some other cases, GraphJIT spends lots of efforts to traverse the whole MHG, but none of MHs is fused in the end. Another significant explanation for the speedup diversity is that bytecodes generated from JRuby tests contain a number of instructions, while both *invokedynamic* and method handles are only two components of them. In other words, compared to the whole test execution time, the changes on the execution time of MHG
traversals would be trivial, if the test is not dynamic method invocation intensive. Together with Figure 6.19b, the mean speedup for those tests with positive values is 15.64%.

<table>
<thead>
<tr>
<th>Speedup</th>
<th>Percentage of Tests</th>
<th>Mean Speedup(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;0</td>
<td>42%</td>
<td>15.63</td>
</tr>
<tr>
<td>=0</td>
<td>19%</td>
<td>0</td>
</tr>
<tr>
<td>&lt;0</td>
<td>39%</td>
<td>-24.23</td>
</tr>
</tbody>
</table>

Table 6.1: Speedup Distribution

This result shows that not all tests can be directly benefit from bytecode generation solution. Users can investigate graph patterns in their own applications, and tune performances by filter configurations, so that GraphJIT can selectively choose transformation nodes for fusion (Refer to Section 6.9.5).

### 6.9.2.2 Evaluation with a common MHG

An MHG, shown in Figure 6.20\(^{11}\), is also built to simulate the structure that is frequently seen in the worst test `cal`. As a JRuby test in the benchmark involves various bytecode instructions (`invokedynamic` is only one of them), this common MHG is created and evaluated, so that the `ExecTime` of the MHG can be isolated from the whole benchmark test execution. For this MHG, another two MHGs are built by GraphJIT with (`with_IC`) and without (`no_IC`) inline caching optimization, respectively. In the experiment, the IC

---

\(^{11}\)The produced figures are in Appendix A.5.
only applies to the node \textit{g1763283466}.

These three MHGs are executed repeatedly by 5000 times. Figure 6.21 illustrates the \textit{ExecTime} comparison. In the figure, the \textit{x}-axis is the number of MHG executions, and the \textit{y}-axis the \textit{ExecTime}.

According to Figure 6.21, the traversal of the MHG produced by GraphJIT without IC outperforms that of the source MHG, and the mean speedup is 31.53%. The gap is large initially, and this gap decreases to be nearly 0, when both MHGs are extremely hot. Besides, the \textit{ExecTime} of \textit{with IC} is close to that of \textit{no IC}, but the former is greater than the latter, when the number of executions exceeds 7*250. This is because the common MHG is constant, and the receiver checking, which is only introduced for \textit{with IC}, an
Figure 6.21: ExecTime comparison for the common MHG from cal (MHG TR JIT threshold is 1000), and Speedup_no_{ic} is 31.53%; 1st JIT completes at about 7*250;
extra work but is not necessary, as it always returns true. These checkings result in a larger code size, and might confuse TR and prevent TR from other further optimizations.

6.9.3 Effectiveness

During evaluation, a verbose log is enabled for both GraphJIT and the TR JIT compiler. GraphJIT logs each graph node it visits and its compilation result, as well as the compilation time expense. Similarly, TR JIT in J9 also logs each method it compiles and the corresponding time expense.

The effectiveness analysis is not applied to MHGs that a) have less than three graph nodes; and b) have three nodes but two leaves. These graphs are too simple for GraphJIT to optimize, and GraphJIT’s compilation effort on these MHGs is a waste of computing resources at runtime.

Based on both verbose logs, the effectiveness of GraphJIT is evaluated from following two perspectives.

The number of Reduced MHs by GraphJIT The percentage of the reduced MHs by GraphJIT is shown in Figure 6.22 for individual tests. This figure shows that GraphJIT is effective in reducing the number of internal graph nodes (i.e., 77.2%).

This measurement strongly depends on the way an MHG is built. In this figure, the varied percentage of reduced MHs leads us to infer that different MHGs have varied internal MHs. This in turn indicates that the ways to
build MHGs for individual tests are also different.

**GraphJIT’s Compilation Expense** Table 6.2 on Page 211 shows the number of MHGs generated by GraphJIT, the percentage of the transformation runtime expenses to the total execution time of a test, and compilation expense per MHG.

According to the table, the mean percentage is 4.58%, and the time expense per MHG is 13.59ms. Considering all these tests are method invocation intensive and each MHG consists of 40 MHs on average, the overhead on GraphJIT is not expensive.
6.9.4 TR Just-In-Time Compilation Impacts

This section studies GraphJIT’s impacts on TR JIT compilation and compilation time. Different from GraphJIT, TR is a JIT compiler that dynamically translates bytecodes into machine code in J9. When GraphJIT is enabled, TR processes two kinds of method handle graphs: a) MHGs that are produced by GraphJIT and b) MHGs that are built by JRuby interpreter, but are not processed by GraphJIT.

An MHG TR JIT compilation consists of two phases. In the first phase (i.e., an MH class JIT), the TR JIT compiler translates each MH class into a native version with little optimization. In the second phase (i.e., an MHG TR JIT), TR inlines these native MH versions of an MHG together with intensive optimizations, when they are frequently executed.

Impacts on TR JIT Compilations  The measurement JITRatio is defined as

\[
\text{JITRatio} = \frac{\text{Num}_{\text{Orig}} - \text{Num}_{\text{GraphJIT}}}{\text{Num}_{\text{orig}}} \times 100\%.
\]

where \(\text{Num}_{\text{Orig}}\) and \(\text{Num}_{\text{GraphJIT}}\) are the number of MH TR JIT compilations for a test without and with GraphJIT, respectively.

Figure 6.23 shows JITRatio for MH TR JIT compilations. According to the figure, GraphJIT increases the number of MH class JIT compilations, and decreases the number of MHG TR JIT compilations by 30.2% on average. For the second JIT compilation, only three of 38 JITRatios are negative, since
more MHGs’ invocation counters in these three tests reach the threshold for the MHG TR JIT.

The reason for increment of JITRatio for the first JIT (i.e., MH class JIT compilation) is that GraphJIT creates a number of MH classes (i.e., new MH transformations) at runtime. In GraphJIT, two MHGs, starting with MHs that have the same class (e.g., `GuardWithTesthandle`) and method type, but different graph structures, would result in two different new MH classes (e.g., `DYNGuardWithTesthandle1` and `DYNGuardWithTesthandle2`). When GraphJIT is disabled, all MHs would result in only one first JIT compilation, if they have the same transformation type and method type, regardless their MHG structures. Thus, TR would see more MH classes for the first JIT when GraphJIT is enabled.

For the MHG TR JIT compilation, the number of MHG JIT compilations is effectively reduced by 30.2%, because internal MHG nodes are fused, and fewer MH instances would be available for TR. In other words, GraphJIT helps reduce MHG TR JIT startups.

**Impacts on MHG TR Compilation Time** Another impact is MHG TR JIT compilation time. TR JIT compilations are separated from the main application thread. Intuitively, this compilation time would increase, owing to MH fusions (e.g., bytecodes of a newly generated node is likely to be more complicated). Our result is consistent with this concept, and shows that this mean compilation time per MHG increases from 34us to 130.04us
Figure 6.23: GraphJIT vs. TR JIT (Mean JITRatio for the 1st and 2nd JIT: -55.5% and 30.2%, respectively)
when GraphJIT is enabled.

6.9.5 Speedup Factors

A test’s execution time can be affected by many factors, such as the number of JIT compilations, global GCs, the quality of the generated code by GraphJIT, among other factors. A key factor is how a dynamic method is called by the bytecodes that are generated by JRuby interpreter. In this section, we analyze the correlation between the execution speedup and the number of TR JIT compilations, the number of MHGs processed by GraphJIT, and filters configured for GraphJIT.

Speedup vs. TR JIT compilations Figure 6.24 shows the correlation between execution speedup and JITRatios for two phases of TR JIT compilation. Based on the figure, the Pearson correlation coefficient\(^\text{12}\) between JITRatio and Speedup for the first JIT is 0.55, while it is -0.03 for the second JIT compilation. We draw two conclusions

- There is a slight correlation between execution speedup and the second TR JIT compilation. When JITRatio is 0.0, the speedups are distributed irregularly between (-20%, 20%).

- The speedup has a positive correlation with first TR JIT compilation.

According to the figure, the distribution of \((\text{JITRatio}, \text{Speedup}) \) pairs

\(^{12}\)The value of coefficient indicates the relationship between JITRatio and Speedup. A value of 1 implies that a linear equation between two metrics, and the value of 0 indicates no relationship.
for the first JITs increases and becomes dense along with the JITRatio, as shown in Figure 6.25. In other words, it is likely to reduce execution time if GraphJIT generates a huge number of MHGs for first TR JITs. The speedup would be clear when appropriate number of MHGs are translated by GraphJIT (e.g., JITRatio ∈ (−20%, 20%))

**Speedup vs. MHGs** Figure 6.26 shows the correlation between the speedup and the number of MHGs that GraphJIT translates. According to the figure, MHGs should be chosen carefully for GraphJIT’s compilation. First, choosing all MHGs for GraphJIT does not help for speedup. As shown in the figure, the increment on the number of MHGs for
Figure 6.25: Figure 6.24's Zoom out when JITRatio is (-0.2, 0.2)
GraphJIT does not contribute to the speedup, when the number of MHGs exceeds 200. The speedup is worsened when the number of MHGs becomes 600 (this is for the test cal). Second, speedups can either be positive or negative, when the number of MHGs for GraphJIT is less than 100. In other words, the compilation of some inappropriate MHGs by GraphJIT does not contribute positively to execution speedup.

**Speedup vs. Filters** As discussed in Section 4.2.1 a transformation pattern is a combination of MH transformation types in a graph; A dynamic JVM language interpreter has preference in the way to combine transforma-
tion types together for a method invocation. In an interpreter, the transfor-
mation patterns to complete a dynamic method might differ from patterns
for other dynamic methods.

A filter in GraphJIT is a graph node’s class name, which represents a trans-
mformation pattern. In GraphJIT, a sub-graph that begins at a node of the
filter type would be skipped from translation directly. Filters, a list of class
names, can be configured for GraphJIT as a command option.

GraphJIT provides filters, instead of original transformation patterns, for
developers to investigate the correlation between transformation types and
the speedup. There are about 20 MH transformation types, and thus in
total \( \sum_{i=1}^{k} 20^i \) (\( k \) is the chain’s length) transformation patterns. As a result,
it is impossible to iterate all patterns in such huge space to investigate the
correlation. Therefore, five frequently seen method handle transformations
are chosen as candidate filters. Each test is repeatedly executed with three
times with only one of the filters.

The correlation between speedup and filters for the three worst tests in Fig-
ure 6.19b is shown in Table 6.3 on Page 212. In this table, the first row
refers to speedup when the filter is empty. From the table, the filter Guard-
WithTestHandle is significant for the test cal and fiber_ring, as the maximal
speedups for both can reach 19.8641% and -24.2254% from -111% and -
73%, respectively, when the filter GuardWithTestHandle is configured. This
huge improvement indicates that some subgraph patterns, beginning with
GuardWithTestHandle transformation\textsuperscript{13} are the bottle neck for performance speedup, and these subgraphs should be excluded from fusion. This might be because the fused bytecodes from these subgraphs are so complex (e.g., number of loops, if conditions, frames) that TR JIT system does not process it effectively. From the table, we can also conclude that the filters that tests are sensitive to are different. For example, both tests \textit{cal} and \textit{fiber\_ring} are sensitive to the filter \textit{GuardWithTestHandle}, while \textit{app\_mandelbrot} is more sensitive to the filter \textit{BruteArgumentMoverHandle} than others.

The correlation between speedup and filters is caused by transformation patterns in MHGs that are built by JRuby interpreter. Some dynamic methods involve intensive \textit{GuardWithTestHandle}s, while others do not. This correlation is also impacted by GraphJIT, since the cost to compile different transformation patterns and the benefit obtained from compiling that transformation pattern vary.

Based on these observations, GraphJIT can be applied to real applications by choosing approximate filters, to maximize performance speedup. In a development environment, users can tune the filter for their applications, and then choose the optimized one, which corresponds to maximal speedup, for their production systems.

\textsuperscript{13}Subgraph transformation patterns can be found in Table 4.1
<table>
<thead>
<tr>
<th>Testid</th>
<th>MHGs</th>
<th>Percentage(%)</th>
<th>Mean expense per MHG (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mergesort_hongli</td>
<td>190</td>
<td>9.23</td>
<td>12.2</td>
</tr>
<tr>
<td>so_object</td>
<td>103</td>
<td>1.11</td>
<td>11.31</td>
</tr>
<tr>
<td>gc_msb</td>
<td>12</td>
<td>2.09</td>
<td>22.25</td>
</tr>
<tr>
<td>socket_transfer_1mb</td>
<td>69</td>
<td>12.12</td>
<td>11.36</td>
</tr>
<tr>
<td>app_factorial</td>
<td>23</td>
<td>3.75</td>
<td>18.69</td>
</tr>
<tr>
<td>string_concat</td>
<td>12</td>
<td>0.26</td>
<td>22.75</td>
</tr>
<tr>
<td>printf</td>
<td>95</td>
<td>4.33</td>
<td>12.58</td>
</tr>
<tr>
<td>so_lists</td>
<td>46</td>
<td>0.83</td>
<td>14.54</td>
</tr>
<tr>
<td>so_lists_small</td>
<td>46</td>
<td>13.58</td>
<td>15.15</td>
</tr>
<tr>
<td>fasta</td>
<td>110</td>
<td>4.13</td>
<td>10.29</td>
</tr>
<tr>
<td>count_multithreaded</td>
<td>33</td>
<td>12.31</td>
<td>28.78</td>
</tr>
<tr>
<td>app_mandelbrot</td>
<td>45</td>
<td>1.37</td>
<td>13.77</td>
</tr>
<tr>
<td>primes</td>
<td>81</td>
<td>8.14</td>
<td>14.97</td>
</tr>
<tr>
<td>nbody</td>
<td>317</td>
<td>2.55</td>
<td>8.35</td>
</tr>
<tr>
<td>app_fib</td>
<td>29</td>
<td>7.83</td>
<td>13.89</td>
</tr>
<tr>
<td>socket_transfer_1mb_noblock</td>
<td>87</td>
<td>20.07</td>
<td>11.97</td>
</tr>
<tr>
<td>nsieve_bits</td>
<td>68</td>
<td>0.82</td>
<td>12.42</td>
</tr>
<tr>
<td>gc_string</td>
<td>25</td>
<td>1.22</td>
<td>16.52</td>
</tr>
<tr>
<td>simple_server</td>
<td>33</td>
<td>7.03</td>
<td>14.0</td>
</tr>
<tr>
<td>gc_array</td>
<td>25</td>
<td>2.55</td>
<td>17.2</td>
</tr>
<tr>
<td>word_anagrams</td>
<td>63</td>
<td>2.94</td>
<td>12.88</td>
</tr>
<tr>
<td>app_pentomino</td>
<td>411</td>
<td>2.80</td>
<td>7.15</td>
</tr>
<tr>
<td>so_sieve</td>
<td>41</td>
<td>1.81</td>
<td>15.29</td>
</tr>
<tr>
<td>eval</td>
<td>11</td>
<td>0.89</td>
<td>20.18</td>
</tr>
<tr>
<td>so_matrix</td>
<td>92</td>
<td>2.04</td>
<td>9.97</td>
</tr>
<tr>
<td>fractal</td>
<td>103</td>
<td>1.24</td>
<td>10.01</td>
</tr>
<tr>
<td>simple_connect</td>
<td>28</td>
<td>7.19</td>
<td>16.10</td>
</tr>
<tr>
<td>partial_sums</td>
<td>79</td>
<td>0.92</td>
<td>12.18</td>
</tr>
<tr>
<td>pi</td>
<td>0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>so_array</td>
<td>42</td>
<td>3.60</td>
<td>12.76</td>
</tr>
<tr>
<td>count_shared_thread</td>
<td>33</td>
<td>4.80</td>
<td>30.12</td>
</tr>
<tr>
<td>fiber_ring</td>
<td>110</td>
<td>9.51</td>
<td>10.43</td>
</tr>
<tr>
<td>list</td>
<td>115</td>
<td>3.19</td>
<td>9.84</td>
</tr>
<tr>
<td>binary_trees</td>
<td>0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>app_tarai</td>
<td>31</td>
<td>0.96</td>
<td>12.61</td>
</tr>
<tr>
<td>cal</td>
<td>611</td>
<td>7.84</td>
<td>7.40</td>
</tr>
<tr>
<td>observ</td>
<td>71</td>
<td>10.28</td>
<td>13.95</td>
</tr>
<tr>
<td>write_large</td>
<td>9</td>
<td>3.32</td>
<td>26.0</td>
</tr>
</tbody>
</table>

Table 6.2: GraphJIT Expense(Mean percentage: 4.58%; Mean Expense per MHG: 13.59ms)
<table>
<thead>
<tr>
<th>Filter</th>
<th>cal</th>
<th>fiber_ring</th>
<th>app_mandelbrot</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>-111%</td>
<td>-73%</td>
<td>-56%</td>
</tr>
<tr>
<td>BruteArgumentMoverHandle</td>
<td>-0.16%</td>
<td>-29.13%</td>
<td>-2.59%</td>
</tr>
<tr>
<td>CollectHandle</td>
<td>14.69%</td>
<td>-29.81%</td>
<td>-9.33%</td>
</tr>
<tr>
<td>GuardWithTestHandle</td>
<td>19.86%</td>
<td>-24.22%</td>
<td>-14.10%</td>
</tr>
<tr>
<td>FilterReturnHandle</td>
<td>5.20%</td>
<td>-28.54%</td>
<td>-35.18%</td>
</tr>
<tr>
<td>ReceiverBoundHandle</td>
<td>5.57%</td>
<td>-34.19%</td>
<td>-17.22%</td>
</tr>
</tbody>
</table>

Table 6.3: Speedups vs. Filters for the Three Worst Tests.
Chapter 7

Conclusions

A Method Handle (MH) is a typed, directly executable reference to an underlying method, constructor, field, or similar low-level operation, with optional transformations of arguments or return values. These transformations are quite general, and include such patterns as conversion, insertion, deletion, and substitution [13, 16, 14]. A Method Handle Graph (MHG) consists of multiple interconnected MHs, and it is a Frequently Traversed but Stable Directed Acyclic (FTSDA) graph.

Both MHs and the instruction \textit{invokedynamic} are introduced in JSR292 to support dynamically typed languages on the JVM. For a dynamic method invocation with the instruction \textit{invokedynamic}, the JVM calls the corresponding bootstrap method, which creates an MHG and links it at the call site, and the linked MHG then transfers the dynamic invocation to real method implementations at runtime.
In this dissertation, we have presented a number of techniques to understand and optimize method handle graphs, to better support dynamically typed languages in the JVM. These techniques are MHG pattern mining, runtime MHG deduplication, and dynamic bytecode generation for MHGs. These techniques will contribute to the integration of dynamically typed languages and the Java Virtual Machine ecosystem, and the productivity and development of dynamically typed languages.

**MHG Pattern Mining**  Two kinds of method handle patterns are identified: transformation patterns (i.e., combinations of method handle transformations) and instance patterns (i.e., equivalent method handle chains). For this technique, we built a Method Handle Mining System (MHMS), in which suffix-trees are built to find frequent MH transformation patterns, and equivalent method handle graphs are detected and grouped for analysis. Based on these data, three of our findings are

- The frequency of the different transformation patterns varies significantly, and a small number of transformation patterns occur much more frequently than the others.

- A large number of equivalencies (i.e., 28.83% of MHs have equivalent MHs on average for JRuby micro-indy benchmark) exist among method handles.
• The distribution of equivalent sets size is also uneven even when the chains in these sets have the same length.

**MHG Deduplication**  This technique includes an MH equivalency model based on MH graph keys, and a runtime MH Deduplication System (MHDeS), which implements the model and avoids creating an MH that would be equivalent to others. As a runtime system, MHDeS speeds up the deduplication procedure and minimizes the performance impact by MHG index keys and a fast-path comparison that detects non-equivalent MHGs as early as possible. Our experiments show that 1) MHDeS can speed up Ruby benchmarks by 4.67% on average, 2) as many as 32% of MHs have been eliminated during the runtime, and 3) memory usage does not benefit much from the deduplication.

**Dynamic Bytecode Generation**  This technique converts an MHG into another equivalent but simpler MHG, and then replaces the original source MHG by the new MHG for execution. The conversion is at the bytecode level (i.e., both source and target are bytecodes), and it is completed by a) fusing graph internal hot MHs, and b) moving leaf MHs close to the root of the graph.

The dynamic MHG bytecode generation is achieved by GraphJIT, which is a JIT compiler for FTSDA graphs, and a method handle template system. The template system generates bytecodes for individual MHs of a graph. These bytecodes, representing individual MHs, are then selectively fused to-

1MHs in an equivalent set are all equivalent.
gether by GraphJIT. The dissertation explains both GraphJIT (i.e., concept, procedures and detailed implementation) and the template system implementation. Furthermore, it demonstrates their integration with the TR JIT compiler for IBM’s JSR292 implementation in J9.

Our evaluation shows that

- the mean speedup for the benchmark is less than 3% when filters are disabled. GraphJIT can speed up 42% of micro-indy benchmark tests by 15.63% on average with maximal value of 67%, while it does not have any impact on execution time for other 19% of benchmarks;

- GraphJIT can speed up a common MHG traversal by 31.53%, when the MHG is isolated from other bytecodes;

- the dynamic bytecode generation solution does not help for garbage collection pause time;

- GraphJIT effectively reduces 77.2% of MHs, while the mean ratio of GraphJIT expense to a test’s execution time is 4.58%;

- the dynamic bytecode generation solution increases the number of first TR JIT compilations by 55.5%, and reduces the number of second TR JIT compilations by 30.2% on average; and

- the dissertation also provides an in-depth correlation analysis between speedup and changes of TR JIT compilations, MHGs compiled by
GraphJIT, and transformation patterns. By selecting a filter care-fully, the speedups for the three worst tests (i.e., cal, fiber_ring, and app_mandelbrot) can be remarkably improved. That means users can maximize an application’s performance by tuning GraphJIT’s filters when the application’s performance is degraded with default configu-

These results indicate that the dynamic bytecode generation can potentially speed up dynamic JVM languages, and reduce the overhead of MHG JIT compilations. The work demonstrates the existence of opportunities to opti-

7.1 Future Work

The planned future work is to

- Investigate correlation between performance speedup and MH trans-

formation types, as well as their combinations. By combining various

MH transformation types, we can get various complex bytecodes (i.e.,

complex subgraphs), based on the quantity of frames and branch in-

structions (e.g., jmp) on the bytecodes. By studying and analyzing the

correlation between these subgraphs and the potential speedup benefits,

we may find out combinations that result in a large system degrada-

tion. To maximize the performance speedup, we would suggest users to

217
configure these combinations (i.e., subgraphs) as filters for GraphJIT’s optimization.

- Tune GraphJIT runtime performance. GraphJIT’s runtime overhead is not trivial, and it has performance impacts at runtime. This runtime overhead can be identified and decreased by resolving bottlenecks during runtime compilation.

- Extend GraphJIT for non-FTSDA graphs. The current GraphJIT detects FTSDA graphs, the structures of which are assumed to be immutable in the most times, and performs optimization automatically. A planned work is to detect and split a graph into multiple FTSDA sub-graphs, if mutable structures are very common.
Bibliography


[2] ASM Type Internal Name, http://asm.ow2.org/asm50/javadoc/user/ org/objectweb/asm/Type.html#getInternalName--


[21] *The Baseline Compiler Has Landed*, [https://blog.mozilla.org/javascript/2013/04/05/the-baseline-compiler-has-landed/](https://blog.mozilla.org/javascript/2013/04/05/the-baseline-compiler-has-landed/).


232


Appendix A

Appendix

A.1 Sample JSON

```json
1{
2   "app_factorial": {
3     "jruby": {
4       "argString": "6000",
5       "elapsed": [
6          0.792,
7          0.797, ...
8      ],
9       "elapsedTime": 15.11732006072998,
10      "footprint": [
11          3.6904220581054688,
12          2.322601318359375, ...
13      ],
```

236
"free": [
  0.5595779418945312,
  1.927398681640625, ...
],
"logPath": ".\tmp\2016−02−01−10.52.05\ruby\app.factorial",
"maxMem": 14.183586120605469,
"status": 0,
"timestamp": "Mon, 01 Feb 2016 15:09:10 +0000",
"total": [
  4.25,
  4.25, ...
]
],
"jruby_optimized": {
  "argString": "6000",
  "elapsed": [
    0.828,
    0.835, ...
  ],
  "elapsedTime": 14.756814956665039,
  "footprint": [
    4.290794372558594,
    2.8044815063476562, ...
  ],
  "free": [
    0.20920562744140625,
    ...}
Listing A.1: Method Handle Transformation Sample

In the sample JSON, it contains two measurement versions for the test `app.factorial`, i.e., `jruby` and `jruby_optimized`. The version with `jruby_optimized` is obtained when the introduced optimization is applied while the version with only `jruby` is without any optimization.

### A.2 Plugin Interface

```java
public interface IPlugin {
    public MethodNode map(String className, String methodName, String desc, MethodContext context,
```
MethodVisitor mv);

public boolean postProcess(MethodNode node, MethodVisitor mv);

public Blob transform(Object receiver, boolean force);

public boolean track(String desc);

}

Listing A.2: Plugin interface in GraphJIT

A.3 Blob definition

public class Blob {

    private Object _generatedObject; //This one should be null

    private MethodNode _methodNode = null;

    private Class _cls = null; //May be not null

    private ClassNode _classNode;

}

Listing A.3: Blob Definition
A.4 GraphJIT and Template System Full Configurations

The base Java version for GraphJIT is 7 if it is Oracle hotspot, and is 8 if it is IBM J9. The template system is only compatible with IBM J9. The following JVM command options can be applied by \(-Dkey=value\) (e.g., \(-Ddump=true\))

1. **dump** Instruct GraphJIT to dump all transformed graph node after generation. It is false by default.

2. **template_dump** Instruct GraphJIT to dump bytecodes from template generation. It is false by default.

3. **validate_bytecode** Instruct GraphJIT to validate bytecodes after generation. It is false by default.

4. **jit_bytecode** Enable GraphJIT if it is true. This is the top level configuration, and all other options are invalid if this option is set false. It is true by default.

5. **skip_selection** Force GraphJIT to find graph nodes for compilation. It is true by default.

6. **generator** Force GraphJIT to create a specified generator, which is associated with the enabled plugin. Current GraphJIT only supports method handle generators and simpler generators. It is MethodHandle.class by default.
7. **enablecache** Enable cache for bytecodes for graph internal nodes. It is true by default. The selection of cache repository then is determined by another three options **globalcache**, **cache_per mh**, and **cache_per_class**.

8. **interval** Cache purge interval. It is 2 by default.

9. **threshold** An threshold for cache purge. An bytecode resource will be removed if its invocation count value exceeds this value. It is 0.8 by default.

10. **globalcache** A global cache for bytecodes is enabled if it is set. It is true by default.

11. **globalcacheSize** The maximal entries that the global cache can host.

12. **globalcachRatio** The purge thread will purges **globalcacheSize**(1-ratio) entries, if **globalcache** is true.

13. **cache_per mh** Enable **cache_per mh** if it is set true and global cache is set false. It is false by default.

14. **cache_per_class** Enable cache_per_class if it is set true and global cache is set false. It is false by default.

15. **cache_template** Enable internal cache in the template system if it is set. It is false by default.

16. **enable_logger** Enable logger system if it is set. It is false by default.
17. **syn** Force GraphJIT to generate bytecodes asynchronously if it is set false. It is true by default.

18. **debug** Force GraphJIT add debug instructions for bug investigation if it is true. It is false by default.

19. **normalclassloader** Force Loader Service create anonymous class loaders if is set false. It is false by default.

20. **interpreterGenerated** This option is only valid with MH plugins. If set, the *equivalent* child of the newly generated MHG will be a direct handle that references the generated transformation method. Otherwise, it will be the source MHG of the target MHG. It is true by default.

21. **dumpOriginMH** This option is only valid with MH plugins. If set, GraphJIT will dump MHGs (both source and target MHG) to a dot file. It is false by default.

22. **forceAllStaticFields** If set, GraphJIT tries to force all fused fields to be static and final for JIT convenience. It is true by default.

23. **dot_dir** Instruct GraphJIT the place for dot files. This option should be used together with **dumpOriginMH** option. It is /tmp/dot/ by default.

24. **aggressiveinlining** Instruct GraphJIT to merge some branches to minimize local variables during fusion. It is true by default.
25. **JSR292graph** This option is only valid with MH plugins. It forces all generated bytecodes to be inside of `java.lang.invoke` package. It is true by default.

26. **filters** A number of class names, separated by “,”. During translation, a node will be excluded if its class type is contained in this option. It is “” by default.

27. **addCount** If set, GraphJIT will profile each field. In this case, a count of a field will increase by one if the field is executed. It is false by default.

28. **inlinecache** If set, the template will perform inlining cache optimization on a call site at the bytecode level. It is false by default.

### A.5 Produced MHGs for the built MHG
Figure A.1: Produced Figure without IC Optimization for the built MHG
Figure A.2: Produced Figure with IC Optimization for the built MHG
Vita

Full Name: Shijie Xu

University attended:
MS, Tsinghua University, 2007
BS, Northwest University, 2004

Publications


Disclosures

- Shijie Xu, Daniel Heidinga, and David Bremner. Method Handle Just-in-time compilation from bytecode to bytecode for efficient native just-in-time compilation. CA8-2016-0002.