8-2010

JVM-based Techniques for Improving Java Observability

Peng Du
University of Nebraska at Lincoln, pdu@cse.unl.edu

JVM-BASED TECHNIQUES FOR IMPROVING JAVA OBSERVABILITY

by

Peng Du

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Witawas Srisa-an and Matthew Dwyer

Lincoln, Nebraska

August, 2010

JVM-BASED TECHNIQUES FOR IMPROVING JAVA OBSERVABILITY

Peng Du, M. S.

University of Nebraska, 2010

Adviser: Witawas Srisa-an and Matthew Dwyer

Observability measures the support of computer systems to accurately *capture*, *analyze*, and *present* (collectively *observe*) the internal information about the systems. Observability frameworks play important roles for program understanding, troubleshooting, performance diagnosis, and optimizations. However, traditional solutions are either expensive or coarse-grained, consequently compromising their utility in accommodating today's increasingly complex software systems. New solutions are emerging for VM-based languages due to the full control language VMs have over program executions. Existing such solutions, nonetheless, still lack *flexibility*, have high *overhead*, or provide limited *context information* for developing powerful dynamic analyses.

In this thesis, we present a VM-based infrastructure, called *marker tracing framework* (MTF), to address the deficiencies in the existing solutions for providing better observability for VM-based languages. MTF serves as a solid foundation for implementing fine-grained low-overhead program instrumentation. Specifically, MTF allows analysis clients to: 1) define custom events with rich semantics ; 2) specify precisely the program locations where the events should trigger; and 3) adaptively enable/disable the instrumentation at runtime. In addition, MTF-based analysis clients are more powerful by having access to all information available to the VM.

To demonstrate the utility and effectiveness of MTF, we present two analysis clients: 1) *dynamic typestate analysis* with *adaptive online program analysis* (AOPA); and 2) *selective probabilistic calling context analysis* (SPCC). In addition, we evaluate the runtime performance of MTF and the typestate client with the DaCapo benchmarks.

The results show that: 1) MTF has acceptable runtime overhead when tracing moderate numbers of marker events; and 2) AOPA is highly effective in reducing the event frequency for the dynamic typestate analysis; and 3) language VMs can be exploited to offer greater observability.

Acknowledgement

I am deeply grateful to Witawas Srisa-an for supporting and mentoring me throughout my master studies. Dr. Srisa-an has provided invaluable technical expertise and feedback in all projects we have taken. Dr. Srisa-an kindly accepted me into his group and guided me enter the field of computer system research. He is always enthusiastic and positive about research ideas and projects, and teaches me research and writing skills. It has been a great honor and pleasure for me to conduct research under his supervision.

I am also indeted to Matthew Dwyer, co-advisor of my master project. I am thankful to his guidance, support and encouragement. Dr. Dwyer not only originated the ideas in this research, but also offered invaluable expertises and insights. My research has benefited greatly from his enthusiasm, patience, high standards, and depth and breadth of knowledge. Without Dr. Dwyer's original and critical thinking, efforts, and encouragement, this project would not have been possible.

I would also like to thank my defense committee member, Xue Liu for his support on my thesis work. Dr. Liu provided invaluable feedbacks when I presented the preliminary results in his class. My thanks also go out to Ashok Samal, Leen-Kiat Soh, and the other members of the CSE faculty who inspired me to pursue this academic adventure.

I am thankful to have Xueling Chen, Du Li, Xueming Wu, Xiangnan He, Yujun

Wang, Yuanming Liu, and Jinjin Liu as friends. They are always by my side and help me overcome all the hardship being far away from home.

This thesis would not have been possible without the open-source HotSpot JVM. I am indebted to all HotSpot developers for their generous efforts maintaining and improving the HotSpot JVM. The `hotspot-dev` mailing list and the entire HotSpot community have been great resources for solving various technical problems I came across during this project.

I am thankful for the unconditional love, support, and encouragement that my parents have been providing throughout my life. I dedicate this thesis to my grandfather, who passed away last year but sadly was not able to have last words with me. I love you all!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Observability represents the level of support inside the computer systems to accurately *capture*, *analyze*, and *present* (collectively *observe*) [49] the internal information, e.g., data structures and program states, about the system. Observability tools assist program developers to understand the code, troubleshoot problems, diagnose performance bottle-necks, and perform optimizations. Traditional observability solutions including assertions, print statements, and debuggers, are useful in many cases. For example, most debuggers support examining program variables, machine registers, and current call stacks at breakpoints. Hand-crafted print statements are more convenient where the state cannot be easily captured by a breakpoint. However, such solutions are either expensive and coarse-grained (debuggers), or static and intrusive (print statements and assertions), consequently defeating the utility of such solutions in accommodating today's increasingly complex software systems.

The introduction of *Virtual Machine* (VM) based languages, e.g., Java and C#, have improved software observability by offering new infrastructures. For example, *Java Virtual Machine Tool Interface* (JVMTI) [2] is a comprehensive Application Programming Interface (API) for implementing analysis clients that can inspect the

internal states of the JVM and control the execution of Java programs. Moreover, the *DTrace* support [1] built into the *HotSpot Java Virtual Machine* also provides a collection of event probes for low overhead analysis clients. The *.NET Profiling API* [59] offers similar features as JVMTI for .NET languages. *Aspect-Oriented Programming* (AOP) can also be used to systematically instrument programs with custom operations.

The proliferation of such instrumentation APIs for VM-based languages can be attributed to the virtualized environments in which the programs are executed. Since native languages, e.g., C and Fortran, execute directly on the bare machine, observability for these languages requires operating system (OS) and hardware support, which are rarely available, if practical at all. Conversely, it is much easier and superior to extend a language virtual machine for observability, because such VMs not only provide all the core services to the programs, but also manage their complete executions. For example, a VM knows the status of every lock being used because programs rely on VMs to perform all locking-related operations.

Although the existing instrumentation APIs have been proved to be useful [12, 46, 61, 68, 10] , they are still limited for implementing fine-grained and more sophisticated analysis clients for the following reasons:

Event Types. For better modularity and ease of programming, most of these infrastructures hide the details of the virtual machines by providing an API for writing program instrumentation. Despite the comprehensive list of supported event types, they only represent a subset of events that developers may want to observe. For example, a developer may want to observe each time a program enters or exits a particular loop. To do so, the developer needs to create an analysis client that traces the entry and exit of a loop in the program and invoke the corresponding callback

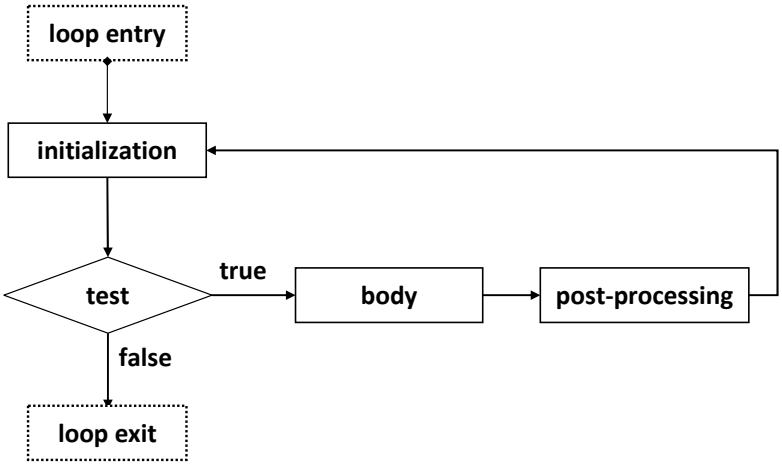

Figure 1.1: Tracing the entry and exit of a loop in the program.

handlers, as shown in Figure 1.1. However, no existing tools support such an event. Furthermore, existing tools do not support analysis that can specify which particular loop to be traced[1]. This type of analysis needs instrumentation support at the *basic block* level, whereas most existing APIs provide instrumentation support at the method level.

Event Filtering. To minimize the space and time overhead, analysis clients require the ability to filter out uninterested event occurrences from the interesting ones (e.g., *invoking the handler only when method A or method B is called, while skipping any other method calls*). Most existing APIs indeed allow monitoring method entry and exit events. Nonetheless, only few techniques, e.g., DTrace, supports user-defined filters for excluding the unwanted occurrences. Without such feature, the instrumentation can cause excessive event notifications because all events are notified but filtering is only performed on the *client side*. When monitoring highly frequent events, client-side filtering incurs significant overhead due to the frequent context switches, where most

[1]JVMTI supports bytecode instrumentation (via 3rd-party tools), which can be used to insert custom code around loops to capture such event.

VM operations must be stalled. Moreover, DTrace filters unwanted events by *polling* all events, resulting in high runtime overhead.

Context Information. Most existing APIs expose various VM internal information by means of a comprehensive set of utility routines. By invoking such routines, analysis clients can query a wide range of context information about most aspects of the program execution, e.g. thread status, current stack trace, object monitor usages, and reachable heap objects. Nevertheless, it is still infeasible to provide specific context information for all analysis clients in the real world. For example, an object allocation and lifetime analysis needs to keep track of the sites where each object is allocated, i.e., *allocation sites*. Although existing APIs support tracing object allocations and reclamation events, they neither identify nor track the corresponding allocation sites.

1.1 Approach

In this thesis, we present an infrastructure that resides in the VM to address the aforementioned deficiencies in the existing instrumentation APIs. The objective is to provide better observability for VM-based languages with a flexible and generic framework for implementing finer-grained low-overhead analysis clients.

1.1.1 Observability by Exploiting Virtual Machine Information

High-level language virtual machines (we will refer to them as virtual machines in the rest of the thesis) are software systems supporting the execution of managed languages, e.g., Java and C#. Because virtual machines serve as the complete execution environments, they can generate rich runtime information during program

execution. In addition, a language VM is a powerful infrastructure by itself, with many useful facilities, e.g., garbage collectors and Just-in-Time (JIT) compilers. Past studies have exploited runtime information and VM facilities to make programs execute more efficiently [37, 75, 76]. In this work, we demonstrate that such exploitation can also lead to improved program observability.

1.1.2 Marker Tracing Framework

We have designed an infrastructure called: *marker tracing framework* (MTF), which supports fine-grained program tracing by allowing analysis clients to define custom events and register handlers. MTF is a generic framework that is completely agnostic of the semantics and handling routine of each user-defined event. Besides fine specification granularity, MTF is also designed to be light-weight, i.e., low runtime overhead and non-intrusive. MTF provides the facilities inside the VM to perform essential operations for analysis clients, e.g., class loading, callback and context management, event dispatch, and JIT compilation.

Analysis clients define custom events by implementing new *markers* — special language constructs for flexibly specify instrumentation points and scopes. A marker consists of an event identifier and meta-data. Each marker encapsulates an arbitrary code region in the program. Analysis clients instrument the programs with markers at compile time. At runtime, MTF can look up and invoke corresponding handler based on the marker identifier for each marker occurrence. Multiple analysis clients can execute in parallel with no interference, even if they register common markers.

Being inside the VM, analysis clients have access to all information available to the VM by implementing *context providers* — specialized VM modules for collecting the needed runtime information by the analysis clients. Analysis developers can

reuse existing context providers or implement new providers when necessary. Context providers are mostly generic and reusable such that we can mix and match context providers to create sophisticated runtime analysis.

MTF can mitigate the fixed-event-type problem in existing APIs by allowing analysis clients to define custom tracing related events. For example, an analysis client can trace every loop in the program by wrapping it with a marker, which then signals the MTF at the loop header and exit respectively. The markers also serve as fine-grained filters in the program where instrumentation is activated; whereas, the rest of the program executes at the full speed. As an example, we can implement an enhanced *Probabilistic Calling Context* (PCC) analysis [20] that can selectively compute PCC values only for the user-specified methods. Because MTF-based analysis and context providers are built inside the VM, they can leverage the VM services to access any context information at runtime.

Although modifying a VM is non-trivial and non-portable, we argue that the benefits with this solution (e.g., access to VM-only runtime information, efficient instrumentation execution, less context switches) far outweigh the disadvantages in many scenarios for enhanced observability.

1.1.3 Effectiveness of Marker Tracing Framework

To demonstrate the utility and versatility of the framework, we have developed two distinct analysis clients, i.e., dynamic typestate analysis with *Adaptive Online Program Analysis* (AOPA) [27] and selective probabilistic calling context analysis [20]. We evaluate the runtime performance of both MTF and the typestate client.

MTF and the analysis clients are implemented for the Java language on the HotSpot JVM. However, we believe the principles and methodologies should apply equally well

for other VM-based languages with comparable effectiveness and runtime performance.

1.2 Contributions

In this work, we present Marker Tracing Framework (MTF), a novel technique offering superior flexibility for the development of dynamic analysis clients for VM-based languages. MTF and the analysis clients collectively improve observability for programs written in these languages. Below we summarize the contributions made by this work:

1. We identify the deficiencies in the current state-of-the-art observability infrastructures: 1) statically fixed event types and programming interfaces; 2) lack of framework-side event filtering; and 3) insufficient context information for some analysis clients.

2. We describe the methodologies behind MTF in detail. We also present an prototype implementation of the framework, which is both efficient and extensible, hence providing a solid foundation for future work.

3. We demonstrate the usefulness and effectiveness of MTF by implementing three instrumentation clients: dynamic typestate analysis and selective probabilistic calling context analysis. Thorough performance evaluations of both the framework and the clients are conducted and presented in the thesis.

4. We show how to leverage internal VM facilities for building custom infrastructures in a production quality JVM with very large code base.

1.3 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 describes the concepts and technologies relevant for understanding the MTF methodology, followed by a discussion of the related work. Chapter 3, 4, and 5 describes the design, implementation, and evaluation of the MTF framework, typestate analysis client, and selective PCC client respectively. We discuss the future work to be explored in Chapter 7 and finally conclude the thesis in Chapter 8.

Chapter 2

Background

To increase robustness, execution efficiency, portability, and standard compliances, most modern JVMs have become very complex (e.g., the current version of Sun HotSpot VM contains over half a million lines of code). Furthermore, applications running on these VMs have also gained complexities over the past few years as developers attempt to exploit thread-level parallelism available in modern multicore processors. As such, it has been challenging for developers to observe the operation of JVM internal mechanisms and structures and understand how these complex runtime systems interact with their applications.

Past studies have shown that leveraging rich runtime information buried inside the JVM can provide developers with the necessary insights to improve the quality and performance of their software. As such, we believe that leveraging such information far outweights the technical complexities to develop means to obtain the information. Therefore, we have designed and implemented the observability framework inside the HotSpot JVM, a production strength JVM that is widely deployed in commercial settings. To facilitate the understanding of the design decisions, this chapter discusses the relevant JVM technologies in general and HotSpot in specific; we leave the

background information on each instrumentation client to the individual chapters.

2.1 Java Virtual Machine

Java is designed to be a high level language executed on top of a virtualized environment — Java Virtual Machine. Such extra abstraction layer is the foundation behind Java's philosophy of portability — "*Write once, run everywhere*". Therefore, Java programs are not translated into platform-dependent machine instructions as programs written in native languages are, e.g., C or C++, but into *Java Bytecodes* [47], a virtual and platform-neutral instruction set suitable for execution on the JVM.

As the name suggests, each bytecode is one-byte in length with an opcode part specifying the operation to be performed, followed by zero or more operands describing the values to be operated upon. Although one byte can be encoded to represent 256 distinct patterns, not all are defined as bytecodes. Similar to existing instruction sets, e.g. x86 and SPARC, Java bytecodes are defined to perform common operations essential for most modern architectures, e.g., memory load and store, arithmetics, object creation and manipulation, control transfer, stack management, and synchronization.

Java virtual machines support automatic memory management based on the concept of garbage collection (GC) [74]. The memory area where Java allocates from and release memory to is called *heap*. Programmers do not explicitly reclaim memory as they do using traditional languages; whereas, the garbage collectors can automatically search for objects that are no longer useful and reclaim the memory. This mechanism effectively release the burden of memory deallocation from the programmers as well as mitigating a host of memory errors, e.g., *memory leak*. Modern JVMs typically incorporate several GC algorithms to suit different requirements. Garbage collectors are sophisticated systems that can provide highly useful information for program

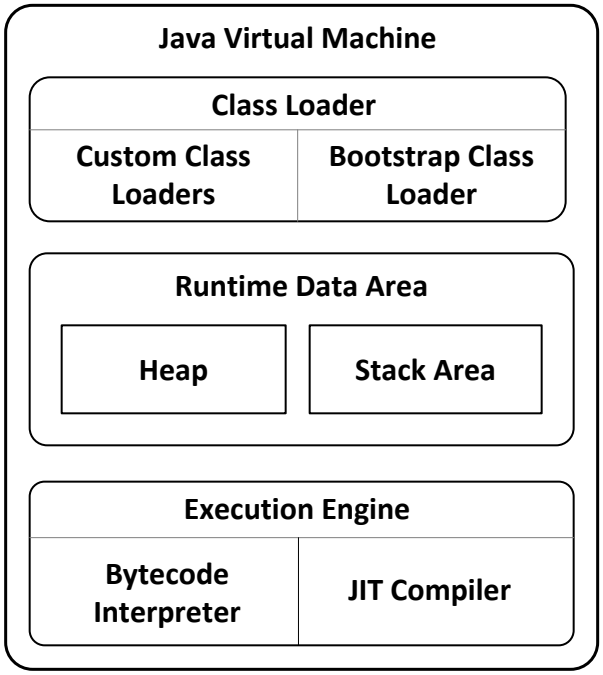

Figure 2.1: Overview of the architecture of a JVM.

instrumentation, i.e., allocation and object death. Figure 2.1 gives an overview of the architecture of a typical modern JVM.

2.2 Class File Format

Java classes, once compiled into bytecodes, are represented by a hardware- and operating system-independent binary format, known as the *class file* format, similar to native object formats like *Executable and Linkable Format* (ELF) [69] and *Common Object File Format* (COFF) [51]. A class file not only contains the bytecode representation of the class but also carry auxiliary information such as constant values, exception tables, and access flags, as shown in Figure 2.2.

Class loaders are responsible for loading and parsing the class files. They create in memory representations of each integral part of the class, e.g., methods, fields, and

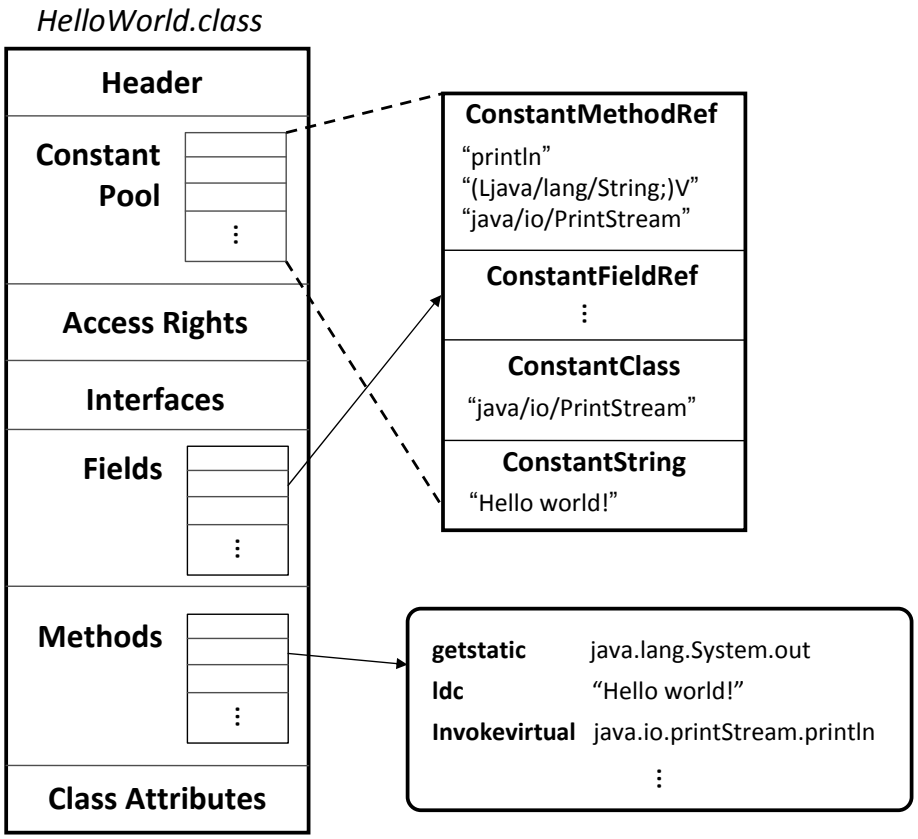

Figure 2.2: The structure of a Java class file.

constant values, for the JVM as shown in Figure 2.3. Besides the *bootstrap* class loader provided by the JVM, programmers can also define custom class loaders to extend the manner in which the JVM dynamically loads and creates classes. For example, instead of loading a class stored in a disk file, we can supply a custom class loader that generates the class on-the-fly directly in memory. In this work, we extend the bootstrap class loader to parse information related to our framework.

2.3 Java Execution Model

Because Java bytecodes are platform-independent, they cannot be executed directly on native processors but are *interpreted* by a specialized execution engine in the

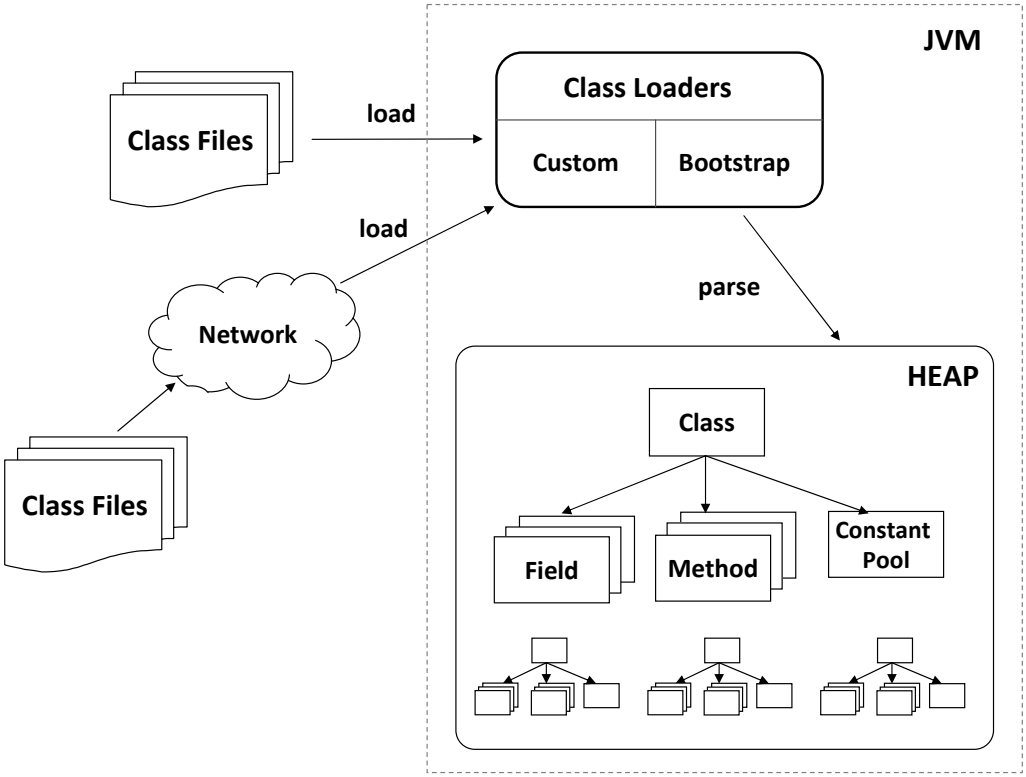

Figure 2.3: Class loaders in a JVM.

JVM, called *bytecode interpreter*. Akin to a microprocessor, the bytecode interpreter performs the typical instruction cycle, i.e., fetching, decoding, and executing Java bytecodes. Figure 2.4 gives a simplified illustration of the execution process of a JVM interpreter. Java is a stack-based language such that computations are carried out on the *expression stack*, i.e., bytecodes pop operands off and push results back onto the expression stack. The interpreter only processes simple bytecodes, e.g. `pushi` and `iadd`, but delegates the complex ones, e.g., `new` and `monitorenter`, to related VM subsystems, e.g., heap allocator and object synchronizer.

Besides the interpreters, most high performance JVMs include *Just-in-Time* (JIT) optimizing compilers, which perform dynamic compilation of bytecodes into optimized machine instructions, which can be executed directly on the native processors, leading

```
do {
     fetch an opcode;
     if (operands) fetch operands;
     execute the action for the opcode;
} while (there is more to do);
```

Figure 2.4: A simplified view of the interpreter instruction cycle [47].

to orders of magnitude speed up. A method is a common compilation unit because of its simplicity for profiling, which is the basis for determining the parts of the code to be compiled. Intuitively, the methods with the most invocations or tight loops yield the most performance gain after compiled by the JIT compiler.

Modern JVMs commonly incorporate multiple JIT compilers and invoke the most appropriate versions with respect to the characteristics of the applications. For example, an interactive application needs to remain responsive and is typically short-running. Thus, a JIT compiler with the shortest compilation time is desirable since it causes the minimum distraction to application activities, although it generates less optimized code. Whereas, a highly optimized JIT compiler is suitable for long-running server-like applications because one-time compilation latency is of less concerns.

2.4 Meta-data

Meta-data, when loosely defined, means the data that can describe the aspects of some other data. There are several scenarios in the Java development process meta-data are necessary for various purposes. For example, the JVM expects the meta-data about the Java class, e.g. descriptors of fields and methods, to be present in the *constant pool* [47] of the corresponding class file as defined the JVM specification. Another option is to store meta-data in "side files" that are kept aside with the applications. For instance, Java applications store configurations and internalization strings in *property*

files. Moreover, JavaEE applications require XML-based deployment descriptors for the configuration information of the various assets. With the release of Java 2 Platform Standard Edition (J2SE) 5.0, Java introduces the *Annotation* meta-data facility [33] for annotating Java code from within Java by allowing descriptive meta-data right next to the language element being described. Programmers can decorate a class, method, field, parameter, variable, constructor, and package with custom annotations.

Depending on the purposes, there are several ways to store meta-data in Java programs. Using plain-text property files has the advantage of being human readable and manually editable using text editors. However, it introduces extra files into the workspace and raises maintenance cost. On the other hand, constant pool stores low-level information essential for class loading and program execution, similar to a symbol table for a conventional programming language. Thus, it is suitable for storing information that is only accessed by the JVM. However, the default bootstrap class loader has to be modified to handle custom meta-data. Java annotation is a high level language feature for storing meta-data directly in source code. It eliminates the needs for "side files" and is easy to edit. However, updating annotations requires recompilation such that is is not suitable for direct use by the JVM.

2.5 HotSpot JVM

Sun HotSpot JVM is one of the most widely deployed production quality Java virtual machine in the real world. HotSpot is a high performance and standard-compliant implementation of the Java Virtual Machine specification [47]. HotSpot supports a variety of mainstream platforms, e.g. IA-32, x86-64, and SPARC, and operating systems, e.g., Solaris, Linux, Windows, and MacOS.

2.5.1 Intepreters

There are two interpreter flavors in HotSpot, i.e. *C++ interpreter* and *template interpreter*. As the name suggests, C++ interpreter is written in C++ with minimal assembly code for low-level stack management, which is not accessible in C++. The core of C++ interpreter is a giant switch-like structure that dispatches each bytecode to the appropriate handling procedure. For performance reasons, HotSpot currently invokes the template interpreter in default. Template interpreter is written in native assembly instructions and generated on-the-fly durign VM startup [34] for each bytecode. HotSpot maintains rich profiling information, e.g. invocation and back branching counts, for selectingly the *hot* methods for JIT compilation.

2.5.2 JIT Compilers

HotSpot provides two JIT compilers: the *client* compiler [44] and *server* compiler [55]. The client compiler has faster compilation speed with fewer optimizations. Thus, it is suitable for interactive and short-running applications where responsiveness is of high priority. The server compiler applies more aggressive code optimizations, thereby incurring longer compilation latency. It is suitable for long-running applications where it is worthwhile to trade initial latency for higher code quality. In HotSpot, the JIT compilers are threaded, thus they execute in parallel with the interpreter cycle. Therefore, a method being compiled is kept interpreted until the compilation is finished. Thus, the number of interpretations is non-deterministic and would vary among runs.

2.5.3 Runtime Systems

2.5.3.1 Object Representation

HotSpot stores Java entities, e.g. instances, methods, classes, arrays, and string symbols, in the runtime Java heap. HotSpot represents them using *Ordinary Object Pointer* descriptor (`oopDesc`) and a variety of subclasses. For example, each array object is represented by an `arrayOopDesc`; regular Java objects are represented by `instanceOopDesc`. HotSpot references these entities by thin pointer wrappers — `oop`, implemented as native machine addresses in memory. Thus, `instanceOop` is the pointer to `instanceOopDesc`. Besides the payload data, e.g., object fields, array elements, and constant pool entries, `oopDesc` encodes the information about each heap object in the *object header*. For example, the header of `arrayOopDesc` records the length of the array; the header of `methodOopDesc` stores the access flags of the Java method.

2.5.3.2 Thread States and Safepoint

Like most sophisticated software systems, threads in the HotSpot JVM can be in three execution states, i.e., the *Java* state, *VM* state, and *native* state. The default state is Java state when HotSpot is executing Java bytecodes. At this state, GC is not allowed since the programs are constantly mutating the Java heap. Similarly, other VM operations that might allocate memory or acquire locks are also prohibited in the Java state. To perform such operations, HotSpot has to switch to the VM state. Thus, entries of runtime routines are guarded by state transition prologues, e.g. Java-to-VM and native-to-VM. In VM state, HotSpot can invoke GC when it reaches a *safepoint*, where all threads in Java state are stalled and GC roots are known. Most call sites and runtime routine entries qualify as safepoints. When HotSpot executes JNI methods,

it switches to the native state. Thread state transitions should be avoided whenever possible because some directions are expensive, e.g., native-to-Java.

2.5.4 OpenJDK

OpenJDK [1] is an open-source implementation of the J2SE specification released by Sun in 2006. The majority of OpenJDK is licensed under the *General Public License* (GPL), except for some encumbered components that can only be distributed as binaries. To date, HotSpot is the only open source implementation of a production quality JVM. Thus, we chose HotSpot in OpenJDK as our research JVM in this work for its production quality, high performance, ubiquitous deployment, and open source code base.

2.6 Bytecode Instrumentation

Because Java classes are compiled into bytecodes and stored in binary class files, it is possible to alter the behaviors and structure of a Java class by transforming the bytecodes and the class files respectively. This technique is called *Bytecode Instrumentation* (BCI). For example, we can introduce new fields and methods, rename classes, and add new interface implementations, on an existing class through BCI. There are many BCI libraries and tools in use, e.g., BCEL [24], ASM [30], JavaAssist [23], and SOOT [71].

Recently, ASM has become one of the most popular BCI libraries due to its small footprint, fast bytecode processing, and ease of use. Unlike previous solutions, ASM does not rely on object representations for the various kinds of nodes in the class tree structure or the various kinds of bytecode instructions. Thus, ASM avoids the

[1] OpenJDK and related projects are hosted at http://openjdk.java.net/.

bloatness and is both time and space efficient in encoding and decoding class files. ASM is based on the *Visitor* pattern [32] and supports two APIs for generating and transforming class files: the *core* API provides an event-based representation of classes, while the *tree* API provides an object-based representation similar to BCEL. Core API is useful for context-free transformations, whereas the tree API is suitable for more complex ones. Each event in the core API represents an element of the class, e.g., header, field, method, and instruction. Transforming class files using the core API requires overriding the virtual methods of several interfaces, e.g., `ClassVisitor`, `FieldVisitor`, and `MethodVisitor`.

In this work, we have developed an ASM-based instrumentation utility based on the core API for the MTF framework, which relies on BCI for embedding markers and associated meta-data into class files. The core API is sufficient for such purpose because marker-related BCI only references the constant pool and related methods, thus is context-free.

2.7 Platform

All experiments of this work are performed on a workstation with Intel Core 2 Duo CPU at 2.40GHz and 4GB physical memory running Ubuntu Linux with x86_64 kernel at version 2.6.32. We base our implementation on OpenJDK 1.7 build 80. We configure to build with the template interpreter and server JIT compiler for x86_64/Linux architecture.

Chapter 3

Marker Tracing Framework

In software engineering, tracing is the process of continuously recording and analyzing certain aspects of the program execution. The utility of trace-based techniques have been demonstrated by a large number of previous research works with diverse focuses, e.g. profiling [36, 35], debugging [73, 21, 60], and dynamic program analysis [48, 62]. Naïve program tracing mechanisms can slow down program execution by orders of magnitude and generates massive trace logs. Tracing also introduces code bloatness into the target program because of the extra logic for recording and processing the traced data. In this chapter, we present a JVM-based infrastructure, the core of which is a light-weight and non-intrusive tracing framework – *Marker Tracing Framework* (MTF).

3.1 Motivation

Tracing is a core technique for effective program instrumentation. Researchers and application developers have been using trace-based techniques for solving real world problems with desirable results. A variety of techniques exist for incorporating tracing

logic into Java programs. The most straightforward approach is implementing custom tracing code directly in the target programs. However, this solutions can pollute the source code and the trace code might not be reusable for other projects. A more elegant and flexible solution is using AOP-based utilities for adding tracing into exiting programs. For example, one of the most widely-used AOP implementation — AspectJ [42], supports highly sophisticated schemes for precisely specifying the parts of the program that should be instrumented. Moreover, since the instrumentation code is written in plain Java, programmers can quickly develop effective instrumentation code leveraging all the language features and the standard libraries of Java. AOP dynamically weaves aspects at compile time, thereby the instrumentation code is completely de-coupled from the target program.

Such application-level tracing techniques are incapable for certain kinds of analyses where low-level information is required. For example, resource and memory leak detections for Java need to monitor object deaths, which is only accessible within the JVM. To address such needs, people have been using VM-based approaches, e.g., JVMTI and DTrace because such frameworks provide interfaces for accessing and mutating VM-only data. Arnold et al. [9] summarizes the advantages of using VM-based instrumentation as:

VM only information. Client analyses can access existing runtime information and record new information that is not possible at the language level, such as stealing free bits in object headers, caching data in thread local storage, and re-using existing VM services.

Performance. Profile data from interpreters and JIT compilers can be used to tune the analysis clients.

Dynamic updating. Advanced techniques, e.g. *code patching* and *on-stack replacement* (OSR) can be used to dynamically adjust the instrumentation at runtime.

Deployment. The instrumentation is built inside the JVM, thus the application is free of modifications such that any applications can leverage the framework if run on such JVM.

Although these infrastructures have been shown to be generally useful, they are still limited in certain aspects and can be improved in the following aspects:

Event Types. Existing solutions uses programming interfaces for providing the services while hiding the details of the virtual machines. Although, the interface is comprehensive, it is still impossible to accommodate all requirements from instrumentation tasks in real world. One of the most confining factors is the fixed event types, which limits the capturing of the exact program states.

Event Filtering. Most existing solutions do not support user-defined filters that are evaluated inside the JVM before invoking the callback functions. Thus, the analysis clients need to listen to all event occurrences and discard the uninteresting ones. The resulted context switches can cause significant overhead for the executions. Therefore, it is desirable if the framework can allow programmers selectively choose which events should be delivered and even dynamically adjust them at runtime.

Context Information. Existing APIs give access to VM information by providing sets of programming interfaces. By calling such routines, analysis clients can query a wide range of context information that is available to the VM. However, programming interfaces cannot suit all possible needs of real world program analyses. The capability of frameworks allowing such analyses to obtain the needed information is necessary.

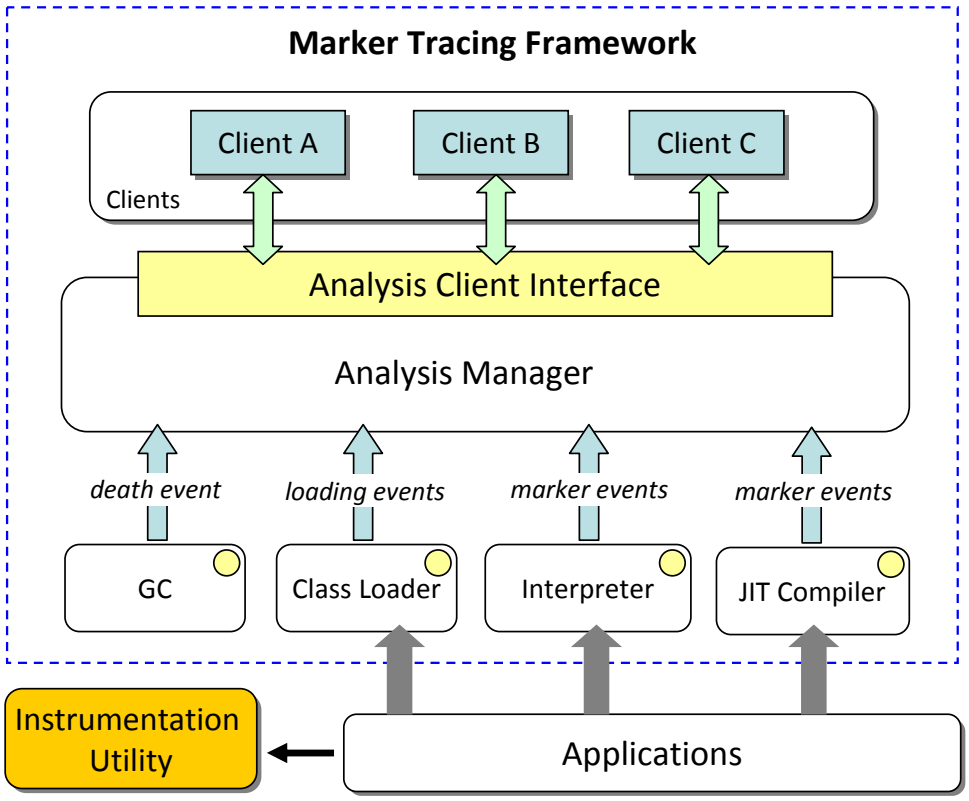

Figure 3.1: Overall architecture of MTF.

3.2 Framework

As shown in Figure 3.1, the MTF framework consists of three parts: *VM Services*, which extend existing VM components and implement the marker bytecodes and tracing mechanisms; *Analysis Manager*, which manages all analysis clients and exposes the underlying VM services to the clients; and *Instrumentation Utility*, a framework for instrumenting programs with marker bytecodes. In this section, we explain the core concepts and sketch the designs of both the analysis manager and instrumentation utility. Because VM services are platform-specific, we discuss them in Section 3.3.

```
public java.lang.Object next();
    Code:
       0: markerenter     2 (mid)
       3: aload_0
       4: getfield        nodeModCount:I
       7: aload_0
       8: getfield        LEDU/purdue/cs/bloat/util/Graph$1;
          ...
          ...
      36: invokeinterface java/util/Iterator.next:()Ljava/lang/Object;
      39: markerenter     1 (mid)
      42: checkcast       java/util/Map$Entry
      44: putfield        last:Ljava/util/Map$Entry;
      47: markerexit      1 (mid)
      50: aload_0
      51: getfield        last:Ljava/util/Map$Entry;
      54: markerexit      2 (mid)
      57: areturn
```

Figure 3.2: An example application of markers.

3.2.1 Marker

The core concept of the framework is *marker*, a special "tag" that can be inserted around a region of code anywhere inside a method. Each marker is represented and referenced in code by an 1-based integer identifier, called *marker id*. We differentiate the entry and exit of a marker (code region) by inventing two new bytecodes into the Java instruction set, i.e., `markerenter` and `markerexit`, similar to the Java monitor synchronizing bytecodes. Both bytecodes take an integer marker id as the only operand. Figure 3.2 shows an example of instrumented with a marker of identifier 2 in a `next` method [1].

Each marker instance is associated with a essential set of context information, i.e., *marker id*, *thread id*, and active *method* and *receiver object*. Such information is necessary for implementing sophisticated analysis clients with thread- and context-

[1]The `next` method is defined by an iterator class inside the `EDU.purdue.cs.bloat.util.Graph` class of the `bloat` benchmark in the DaCapo suite 2006).

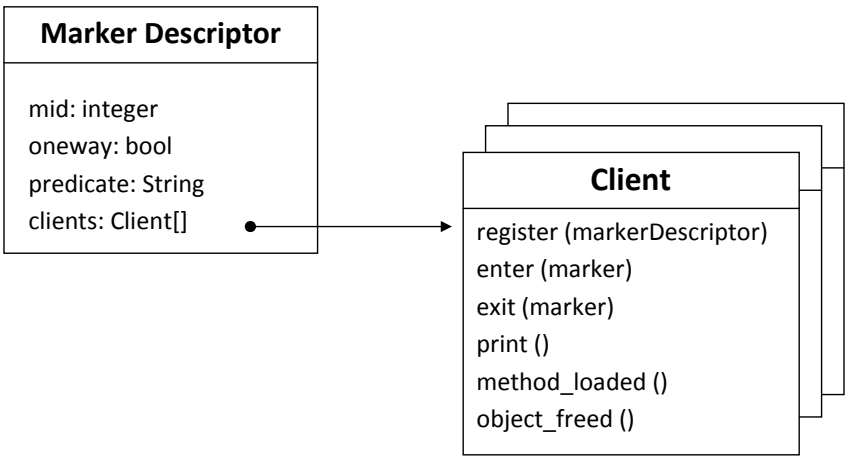

Figure 3.3: The structure of a marker descriptor.

sensitivity. Since the instrumentation clients are implemented inside the JVM, more information can be obtained by exploiting existing runtime services, e.g., interpreter, JIT, and garbage collectors.s

In MTF, a marker can also be without an exit bytecode; such markers are called *one-way* markers, which are suitable when the they are used to signify an event at a specific program location where scope is unnecessary. In this scenario, we can define one-way markers to reduce the overhead resulted from unnecessary context switches.

3.2.2 Marker Descriptor

A *marker descriptor* is the specification of a class of markers that share the same semantics and processing routines, just as *Class* is the template of a set of *Objects* in Object-Oriented Programming. A marker descriptors consists of three parts, i.e., marker id, *predicate*, and set of callback handlers, as shown in Figure 3.3. Marker descriptors are directly embedded in the constant pool of the class file.

The second slot is shared by two string fields: *token* and *predicate*. The token field indicates the kind of the marker for interested analysis clients to recognize and

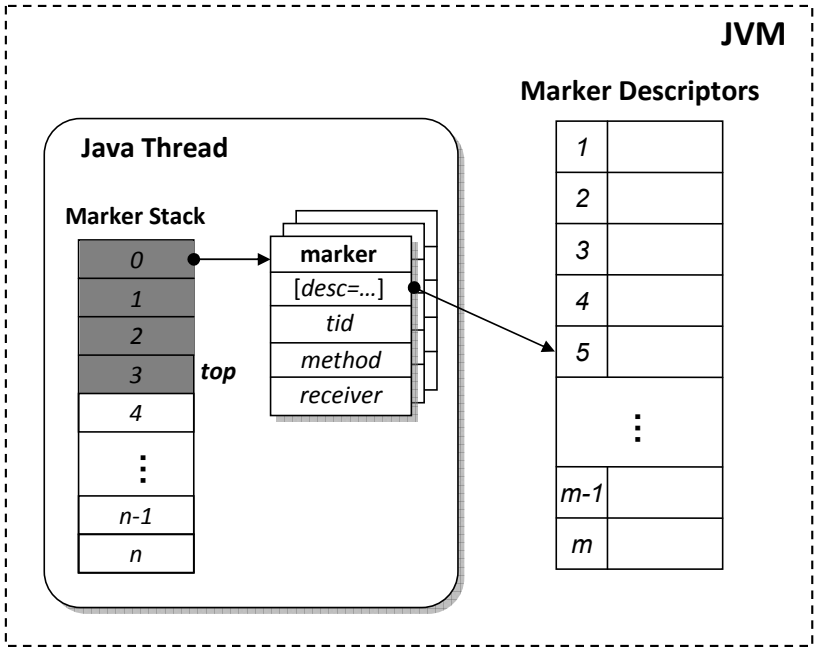

Figure 3.4: Per-thread marker stack

subscribe. On the other hand, the predicate field is optional and is used to specify auxiliary information for the marker handlers to operate upon. Both fields are freely encoded by the clients to properly represent the information necessary for carrying out the analyses.

3.2.3 Marker Stack

Marker stack is a memory buffer for storing marker instances in a *last-in-first-out* (LIFO) order. Each thread has its own marker stack such that the marker push and pop operations are *thread-safe* yet *lock-free*. Figure 3.4 illustrates the per-thread marker stack scheme. Such feature is designed for analysis clients which require context-sensitivity of markers. Walking the marker stack has much lower overhead than the call stack and is more flexible because markers can be inserted in any program locations.

Method	Description
`register`	Registers an analysis client to the framework
`subscribe`	Decides whether to subscribe to a specific marker.
`marker_enter`	Marker enter event callback
`marker_exit`	Marker exit event callback
`object_death`	Object death event callback
`method_loaded`	Callback for processing loaded methods that have markers
`print`	Callback for printing analysis-specific results

Table 3.1: Analysis client interface.

3.2.4 Analysis Manager

In MTF, analysis manager controls and interacts with all analysis clients. For modularity and ease of management, each analysis client must implement an interface defined by the framework, whose members are listed in Table 3.1. Additionally, the communications between the analysis clients and the framework are also based on this interface.

Specifically, analysis clients `register` themselves with MTF and stays on the analysis client list. During class loading phase, MTF parses the constant pools looking for marker descriptors. For each found descriptor, MTF polls the registered clients with the predicate string as an argument. The `subscribe` routine of each client is expected to return a boolean value, indicating whether it wants to subscribe to the events of the marker as specified by the descriptor. When MTF receives a `true` response, it attaches the client to a per-descriptor list that holds all the subscribers. During program execution, when MTF observes a marker occurrence, it indexes into the descriptor list with the marker id and iterates through the list while invoking each handler. MTF creates a `marker` structure with the descriptor and current context information, such as the thread id and stack pointer, and pass the `marker` to the handlers as the only argument.

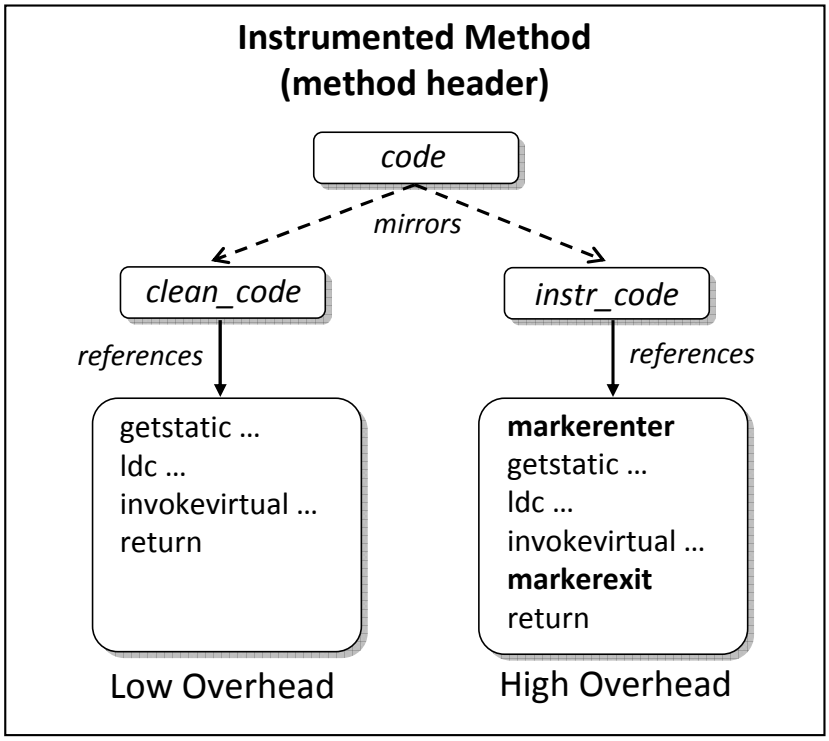

Figure 3.5: Adaptive online program analysis: the MPS approach.

3.2.5 Adaptive Marker Invocation

For allowing analysis clients to adaptively turn on and off marker instrumentation at runtime, we explore two alternative VM-based solutions in MTF focusing on low switching overhead.

Method Pointer Swapping. As described previously, modern JVMs compile Java bytecodes into native instructions for frequently executed methods by using the JIT compiler. Typically, compiled methods are stored in *code buffers* and are referenced by pointers. Once the compilation is finished, the very next invocation to each such method follows the pointer to the corresponding code buffer and starts execution.

Based on this design, *Method Pointer Swapping*(MPS) adds two new pointer, `instr_code` and `clean_code` into the method header alongside the original code

pointer `code`. Moreover, MPS extends the JIT compiler to perform two compilations, including and excluding the marker instrumentation. The resulted code buffers are referenced by `instr_code` and `clean_code`, respectively. During analysis, MPS can adaptively switch between the instrumented and uninstrumented code by assigning either `instr_code` or `clean_code` to `code`, which is the global entry point used throughout the VM. MTF performs such switching on behalf of each analysis client as requested. Figure 3.5 shows Method Pointer Swapping approach.

This scheme is similar to the *Full-Duplication* version of the instrumentation framework presented in [8]. One of the benefits is that the uninstrumented code runs at full speed when no instrumentation is needed. However, it also has several critical disadvantages. First, each compiled method requires two compilations and two code buffers, thus wasting both time and space. Performance is further degraded when JIT compilers perform multiple recompilations to get the best optimization, which is a common optimization in modern JVMs. Second, this technique requires extensive VM modifications to keep the method header and two buffers in synchronization. Based on these drawbacks, we did not use the MPS approach in our final MTF solution.

Marker Dispatch Preamble. To dynamically switch instrumentation code, *Marker Preamble Dispatching* (MDP) injects a tiny piece of check-and-dispatch code preamble for each compiled marker. Unlike MPS, MDP generates only one code buffer per method and the marker bytecodes are always compiled and integrated into the final buffer as shown in Figure 3.6. To achieve the same adaptivity, MDP checks an extra flag in the object header which is accessible by each analysis client. The flag is a bit-mask with fields for all markers. Depending on the state of the bit, MDP either executes or jumps over a particular marker instrumentation. Figure 3.7 shows the control flow of the preamble checking code inserted for each marker bytecode.

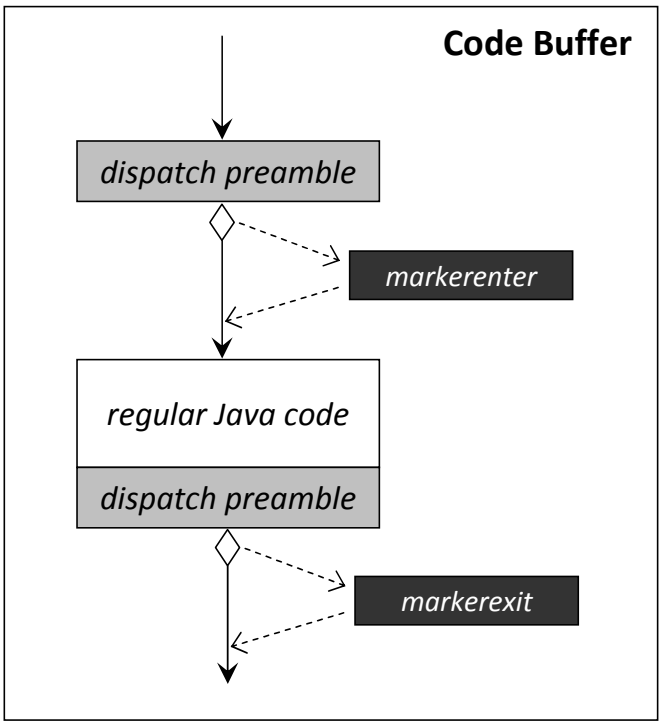

Figure 3.6: Adaptive online program analysis: the MDP approach.

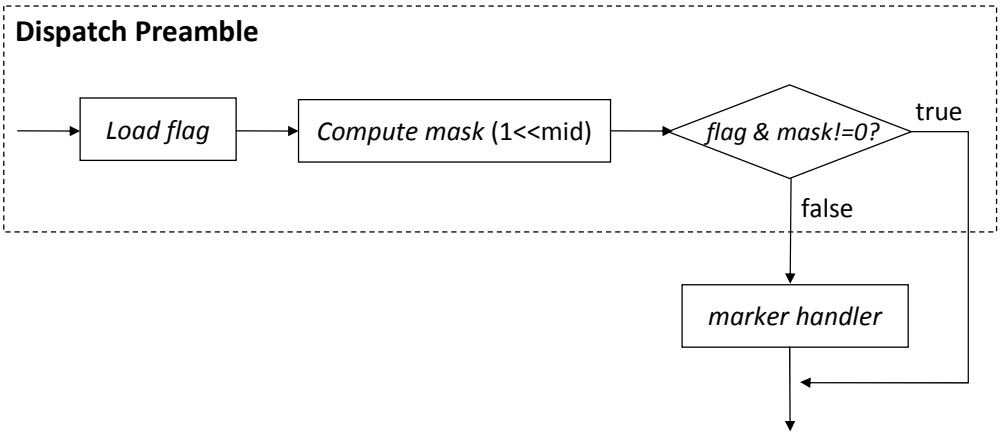

Figure 3.7: Control flow of the dispatch preamble for each marker bytecode.

Method	Description
`getClassVisitor`	Returns the custom class visitor for annotating the bytecodes.
`postProcess`	Optional post-processing step for some instrumentation clients.

Table 3.2: Instrumentation client interface.

MDP has a similar design as the *No-Duplication* approach as presented in [8]. Unlike MPS, MDP requires no extra compilation or code buffer. In addition, the checking code consists of only several machine instructions such that it adds only negligible overhead. Nonetheless, massive event stream can cause a noticeable overhead, although we believe such scenario is very rare in practice. Moreover, the bit-vector implementation does not scale as the complexity of analysis grows when fixed-length vectors are used for simplicity and efficiency. In reality, we expect the number of all active markers in an analysis session to be below 32 or 64, which requires only word- or double-word-length vectors.

Despite the limitations of MPD, we implemented the MPD approach in the final solution for its conceptual and technical simplicity.

3.2.6 Instrumentation Utility

Because standard Java does not specifies the marker bytecodes, i.e., `markerenter` and `markerexit`, none of the existing instrumentation utilities can be used for MTF-based analyses without modification. In this work, we propose a prototype utility based on bytecode instrumentation for annotating programs with marker bytecodes. The utility is designed to be extensible such that new instrumentation clients can be easily developed and added to support new analyses. The utility operates on an input class file and outputs an instrumented copy with the existing bytecodes kept intact such that the original semantics is preserved.

The utility leverages an existing BCI framework, ASM [30], for manipulating

bytecodes stored in a binary class file. Each analysis client is required to implement the `InstrumentationClient` interface (Table 3.2) defined by the utility. In ASM, `ClassVisitor` is the top-level interface that can traverse and transform class-related elements, such as access flags, class name, and constant pool. Specific visitors, e.g., `MethodVisitor` and `FieldVisitor`, are created by the `ClassVisitor` for processing the methods and fields respectively. Each `InstrumentationClient` defines a custom `ClassVisitor` for embedding marker descriptors into the constant pool. A custom `MethodVisitor` is also required for adding marker bytecodes into the appropriate methods.

The main utility performs the I/O of the class files and instantiates the instrumentation client as specified by the user with *reflection*. The utility then traverses the class file with the `ClassVisitor` returned by the client. Consequently, the corresponding instrumentation tasks are executed by the selected client. Once the tasks are finished, the utility saves the instrumented class into a designated class file. Figure 3.8 illustrates the workflow of marker instrumentation as performed by this utility.

3.2.7 Summary

MTF is based on the hybrid approach, i.e., runtime VM-based analysis clients in conjunction with compile-time instrumentation clients, to provide fine-grained analysis-driven tracing functionality. Figure 3.1 shows the overall architecture of the marker tracing framework at a high level. Analyzing programs using MTF involves the following steps:

- characterize the program locations, e.g., method entry, loop header, where the analysis should be notified;
- define the format of the predicate string of the marker descriptor;

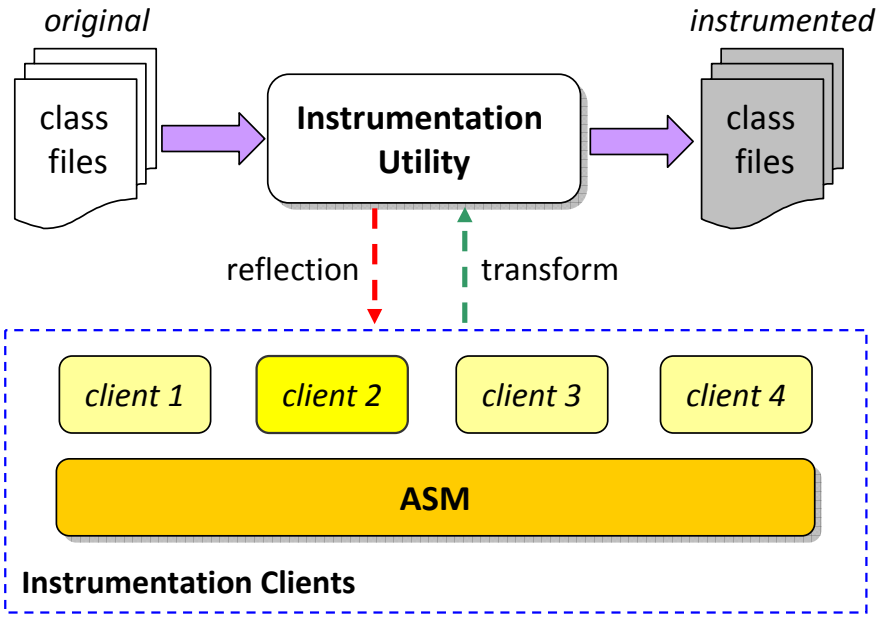

Figure 3.8: Workflow of marker instrumentation.

- insert the markers at those locations and embed the marker descriptors in the constant pool;
- implement and plug in the analysis client in the VM for marker event handling;
- execute the instrumented program on an MTF-enabled VM with the analysis client.

3.3 Implementation

Since MTF extends the Java instruction set and class loading process, several components inside the VM need to be modified for implementing MTF support. Figure 3.1 shows the overview of the architecture of our MTF implementation. This section describes the details of each major component of our implementation in the HotSpot JVM.

3.3.1 Extending HotSpot JVM

Sun implemented HotSpot JVM to support two architectures: x86 and SPARC for both the 32-bit and 64-bit memory models. The majority of the VM is platform independent. Parts of the runtime system, e.g., interpreter, native code generator, and frame manager, are however platform dependent. Although our MTF implementation is OS neutral, it executes only on `x86_32` and `x86_64` because template interpreter and JIT need to be modified for MTF. However, the implementation should be easily portable to other architectures with equal effectiveness.

3.3.1.1 Class Loader

The bootstrap class loader in HotSpot is extended to parse the marker descriptors from the constant pool of each class file. Specifically, the constant pool parsing routine is modified to record the index of the delimiter represented by a 32-bit integer, i.e., `0xbabecafe`, in our implementation. The marker desciptor parsing routine works on entries starting at the index and performs the operations as sketched in Figure 3.9.

The routine checks whether the current class has markers. If such condition holds, it retrieves the marker id, direction (`one-way`), and predicate string (`preds`) from the pool entries. A marker descriptor is then created with the class name and the collected values. Afterwards, it invokes the `accept` method of the `analysisManager`, which iterates over all registered analysis clients with the marker descriptor. The clients can return a `true` value indicating they want to receive notifications on the occurrences of the marker as described by the descriptor. After processing all descriptors, the routine caches the presence of code markers in the `Klass` object of the current Java class. This flag is consulted by the framework for specialized handling in the interpreter and JIT compiler. Analysis clients can also consult the flag if necessary.

```
bool has_markers = marker_index() > 0 ? true : false;
if (has_markers) {
  for (int i=marker_index(); i<length; i+=2){
    // parse marker id
    symbolOop sym = cp->symbol_at(i);
    {
      ResourceMark rm(THREAD);
      buf = sym->as_C_string();
      int len = strlen(buf);
      if (strchr(buf, '*')) { // one-way marker
        oneway = true;
        buf[len-1] = '\0';
      }
      mid = atoi(buf);

      // parse predicates
      sym = cp->symbol_at(i+1);
      buf = sym->as_C_string();
      copy_str(buf, preds);
    }

    markerDescriptor* md;
    if (mid != 0) // regular marker
      md = markerDescManager::add(mid, class_name, preds, oneway);
    else // psuedo marker
      md = new markerDescriptor(mid, class_name, preds, oneway);

    analysisManager::accept(md);
  }
}

Klass* clazz = cp->pool_holder()->klass_part();
clazz->set_has_code_markers(has_markers);
```

Figure 3.9: Marker descriptor parsing routine.

```
void SharedRuntime::markerenter(JavaThread* thread, int mid) {
  pid_t tid = thread->tid();
  intptr_t *sp = thread->last_Java_sp();
  markerStack *st = thread->mkstack();

  // notify predicate watchers
  analysisManager::marker_enter(st->push(mid, tid, sp));
}
```

Figure 3.10: `markerenter` handler in `SharedRuntime`.

3.3.1.2 Shared Runtime

In HotSpot, complex runtime operations, such as slow-path allocation, biased locking, and virtual method resolution, are handled by two underlying runtime systems: `InterpreterRuntime` and `SharedRuntime`. `InterpreterRuntime` serves the bytecode interpreter only, whereas `SharedRuntime` provides services to both the interpreted and compiled code. It is non-trivial to implement the marker routines by either generating assembly templates (interpreter) or constructing intermediate representations (JIT). Thus, we implement the marker routines in `SharedRuntime` such that both interpreted and compiled code can process markers by requesting such services from `SharedRuntime`. Figure 3.10 shows the implementation of `markerenter` as a method of `SharedRuntime`.

As marker context information, `markerenter` gathers the current thread id and stack pointer and pushes the marker on the per-thread marker stack. Then, it notifies the `analysisManager`, which in turn propagates the marker event to the subscriber clients. Other unsubscribing clients are skipped being off the per-descriptor list of the current marker.

Bytecode	Format	Result	Stack	Exception
`markerenter`	`bcc`	`VOID`	0	true
`markerexit`	`bcc`	`VOID`	0	true

Table 3.3: Marker bytecodes definitions.

3.3.1.3 Interpreter

We extend the template interpreter to support the marker bytecodes in HotSpot. In template interpreter, each bytecode is interpreted by an assembly code template generated during VM startup. Except for branching and return bytecodes, the epilogue of each template retrieves the next bytecode and dispatches to the corresponding template. Each template can specify runtime attributes, e.g., operand format, result type, stack effect, and throwing exception. Table 3.3 shows the definition of our marker bytecodes, where "bcc" means one bytecode followed by two bytes (one `short`); `VOID` means no value is returned; 0 indicates the bytecode does not change stack; and `true` indicates the bytecodes might throw exceptions.

The generator of each code template is a C++ method in the `TemplateTable` class, which is implemented specifically for each architecture, i.e., x86_32, x86_64, and SPARC. In HotSpot, low-level code generation is performed by a *macro assembler*, which not only implements the underlying instruction set, e.g., `MOV`, `CMP`, `XOR`, it also provides essential macros for common tasks, such as runtime call, bytecode access, and null pointer check. Assisted by `InterpreterMacroAssembler`, we implemented marker bytecodes as two methods in `TemplateTable` for x86_32 and x86_64. Figure 3.11 gives the implementation of `markerenter` on x86_64. The template starts by loading a two-byte unsigned integer (marker id) at the current `bytecode pointer` (bcp) and saving the value to register `rbx`. Then, it invokes `SharedRuntime::markerenter` with the the marker id in `rbx` as an argument.

```
void TemplateTable::markerenter() {
  transition(vtos, vtos);
  _masm->get_unsigned_2_byte_index_at_bcp(rbx, 1);
  _masm->call_VM(noreg, CAST_FROM_FN_PTR(address,
                 SharedRuntime::markerenter), rbx);
}
```

Figure 3.11: `markerenter` code template generator on x86_64.

3.3.1.4 JIT Compiler

In HotSpot, a Java method is JIT-compiled if it is discovered to be frequently executed or contains a tight loop. To support markers in such methods, we extend the *server* JIT compiler (C2) in HotSpot to compile marker bytecodes into native instructions same as the regular bytecodes.

C2 is based on a *program dependence graph* ("sea-of-nodes") like intermediate representation [56] with several phases, e.g., parsing, optimization, and code generation. To support MTF, we modified the `do_one_bytecode` method of the `Parse` class, which has a comprehensive switch statement for each Java bytecode. We introduced two new labels for parsing the marker bytecodes in the switch statement. Similar to the marker code templates in the interpreter, we generate an equivalent sequence of actions using the compiler's IR. However, we do not have to load the marker id under the current `bcp` as in the interpreter templates, which are generic routines for all marker invocations, because the marker id can be retrieved statically by examining the bytecode stream being compiled. Thus, the compiler can simply generate an integer constant node as the operand for the VM call, e.g., `SharedRuntime::markerenter`.

C2 compiles runtime calls using the Java calling convention, which is different from the C++ convention used by the runtime methods. Thus, C2 generates an adapter code for each runtime method with the proper signature specification. The

adapter code bridges the two calling conventions and eventually makes the runtime call. Specifically, the signature is defined by a "{method}_Type()" method, where "{method}" is the name of the runtime method. Similarly, the adapter code is named "{method}_Java" and the C++ method pointer is named "{method}_C". Thus, we implemented `marker_handler_Type` to provide the same signature for both `markerenter` and `markerexit`.

3.3.2 Instrumentation Utility

Figure 3.12 sketches the algorithm in processing the input class file with the specified instrumentation client (error handling is omitted for brevity). The routine `getBytesFromFile` reads the class file into a byte buffer. To make the `instrument` procedure generic, we instantiate the instrumentation client from the the class name using Java *reflection*. For example, "client 2" is instantiated as the instrumentation to be performed on the program in Figure 3.8. In our utility, the path of the class file, the class name of the client, and the output path are parsed from command line arguments. The utility also supports instrumenting "exploded" JAR files by recursively descending into the directories and process each class file. This feature is particularly useful for batch instrumentation of class libraries with complex package structures and large numbers of classes.

Marker Descriptors

As described previously, constant pool is used to store the marker descriptors for the instrumented class. In ASM, a client can override the `visit` method in `ClassAdapter` to append the marker descriptors to the constant pool. The original entries are separated from marker entries by a delimiter — a magic number (`0xbabecafe`) Each marker descriptor is represented by two entries, i.e., the marker id (`integer`) and

```
public byte[] instrument(File classFile, String className) {
  // Start processing the class file
  byte[] input = getBytesFromFile(classFile);
  ClassReader reader = new ClassReader(input);
  ClassWriter writer = new ClassWriter(reader, 0);

  InstrumentationClient client = (InstrumentationClient)
    Class.forName(className).newInstance();
  ClassVisitor visitor = client.getClassVisitor(writer);
  reader.accept(visitor, 0);
  client.postProcess();

  return writer.toByteArray();
}
```

Figure 3.12: The algorithm of marker bytecode instrumentation.

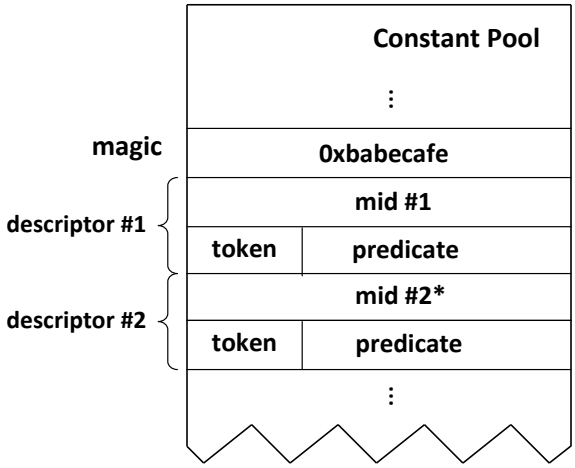

Figure 3.13: Layout of the constant pool after instrumentation.

predicate string (`UTF8`), created by `newConst` and `newUTF8` of the `ClassWriter` object in ASM. Figure 3.13 shows the layout of the constant pool with two marker descriptors embedded with the second one being an one-way marker as indicated by the asterisk.

```
public void visitInsn(int opcode){
  if (opcode >= Opcodes.IRETURN && opcode <= Opcodes.RETURN){
    mv.visitIntInsn(markerexit, <mid>);
  }

  mv.visitInsn(opcode);
}
```

Figure 3.14: Instrument return statements in a method with `markerexit` bytecode.

Markers

Markers are directly added to the methods to be traced at runtime. In ASM, this can be achieved by overriding the corresponding visitor methods in *MethodAdapter*. For example, the client can override the `visitCode` method to add markers at the method entries. Since both marker bytecodes take an integer operand, the `visitIntInsn` in the `MethodVisitor` interface can be used to insert marker bytecodes. For example, `visitIntInsn(markerenter,1)` inserts `markerenter` for marker #1 at the current location being visited. Figure 3.14 shows how to add marker exits to all return statements in the current method, where `mv` is the current `MethodVisitor` object.

Post-processing

In the batch processing mode, it is typical for a client to generate marker identifiers as the instrumentation proceeds. Thus, the number of instrumentation points might not be available before starting. However, the constant pool is visited by the `ClassVisitor` before instructions are visited by the `MethodVisitor`. Furthermore, marker descriptors have to specify marker identifiers in the constant pool when visited. Therefore, our utility provides an interface method `postProcess` for patching the constant pool after all methods have been visited.

Additionally, `postProcess` can return a boolean value, which is checked by the

utility. If true is returned, the utility proceeds to the next class file; otherwise, the whole instrumentation process is aborted. This feature is useful when the client determines it has finished instrumenting all target classes and wants to quit all subsequent traversing altogether.

3.4 Evaluation

In this section, we evaluate the performance of marker tracing framework (MTF). We start by describing the experiment methodology. Then, we report and discuss the experiment results.

3.4.1 Methodology

We choose DaCapo [13] benchmark suite release 09.12 (most current) for this evaluation because DaCapo draws real-world applications with diverse behaviors [2]. Table 3.4 displays the basic characteristics of each benchmark in DaCapo. Column "Executed Methods" reports the total number of application methods (excluding library methods) that are actually executed and recorded by MTF in profiling runs. In the experiments, we exclude the `batik` benchmark because it requires the `JPEG Codec` API which is retired and removed in the OpenJDK 7 code base [3]. For such reason, our JVM cannot execute `batik`.

We evaluate the runtime overhead of MTF by running programs with different instrumentation levels on a "Nop" client, whose `markerenter` and `markerexit` methods perform null operations. Because the overhead of MTF is proportional to the number of events, we randomly sample 10%, 25%, 50%, and 75% of all the executed

[2] see http://dacapobench.org/

[3] see http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6527962

Benchmark	Description	Executed Methods
avrora	Parallel discrete event simulator	1,437
batik	Vector graphics renderer	N/A
eclipse	Eclipse JDT performance tests.	575,590
fop	XSL-FO to PDF convertor.	2,921
h2	Banking application benchmark.	1,501
jython	pybench Python benchmark.	4,610
luindex	lucene-based index generation.	874
lusearch	lucene-based text search.	469
pmd	Java code analyzer.	2,516
sunflow	Rendering system for image synthesis.	583
tomcat	Webpages retrieval and verification against Tomcat.	602
tradebeans	Daytrader benchmark via Jave Beans.	15
tradesoap	Daytrader benchmark via SOAP.	15
xalan	XML to HTML transformation.	2,064

Table 3.4: Basic characteristics of each benchmark in DaCapo 2009 suite.

methods, which are obtained by profiling using MTF. For each sample size, we execute the benchmarks 10 times and take the *arithmetic mean* as the final results. To show the worst-case performance, we also instrument 100% of the executed methods. For comparisons, we run the uninstrumented DaCapo benchmarks on an unmodified OpenJDK with the same version as our MTF-enabled JVM (we refer to this unmodified JVM as *vanilla*). We set the `-converge` option of the DaCapo suite which runs each benchmark multiple times such that the reported execution times are within a confidence interval of 3%.

In the experiments, we only instrument application methods because we believe these methods are more interesting and relevant for the end users to verify, consequently are more representative than the methods from the Java runtime library.

3.4.2 Results and Discussions

In this section, we report the performance of MTF tracing DaCapo benchmark with the `nop` analysis client in Table 3.5, 3.6 and Figure 3.15.

Benchmark	Vanilla	10%	25%	50%	75%	100%
avrora	13,087	**8,642**	16,819	24,892	33,454	43,976
eclipse	41,364	47,547	98,392	153,362	212,275	275,831
fop	446	469	845	1,033	1,452	1,768
h2	7,159	7,345	15,130	23,222	31,842	41,861
jython	3,360	3,521	7,250	10,811	14,487	18,513
luindex	1,039	1,235	2,652	4,108	5,596	7,345
lusearch	3,940	4,679	9,964	15,680	22,561	30,193
pmd	3,808	**3,735**	7,819	12,026	16,619	21,575
sunflow	7,558	9,118	19,504	30,936	43,328	55,111
tomcat	5,448	5,473	10,909	16,510	22,926	29,033
tradebeans	7,639	9,102	19,194	31,479	46,423	61,839
tradesoap	17,088	19,709	42,154	67,216	94,924	123,100
xalan	2,727	3,282	7,250	11,686	17,880	25,037

Table 3.5: Execution time of running DaCapo benchmarks on MTF.

Benchmark	10%	25%	50%	75%	100%
avrora	201	13,420,619	42,872,995	90,676,427	543,934,712
eclipse	338	47,255	104,674	211,352	358,208
fop	1,775	183,711	714,998	1,925,066	10,755,787
h2	68,335,094	155,329,573	251,138,862	350,215,459	1,325,200,083
jython	223	244,681	5,867,601	21,013,345	356,096,651
luindex	159	9,845	22,598	45,680	62,876,224
lusearch	23	20,420,789	81,029,620	284,642,316	886,577,763
pmd	809,858	1,948,717	3,152,651	6,948,307	91,662,920
sunflow	300	1,263	2,885	2,842,068	239,384,047
tomcat	29	136	7,152	803,893	3,795,574
tradebeans	1	4	11	22	39
tradesoap	1	5	11	23	45
xalan	84,693	15,457,628	63,626,175	147,161,490	367,424,093

Table 3.6: Marker invocations of running DaCapo benchmarks on MTF.

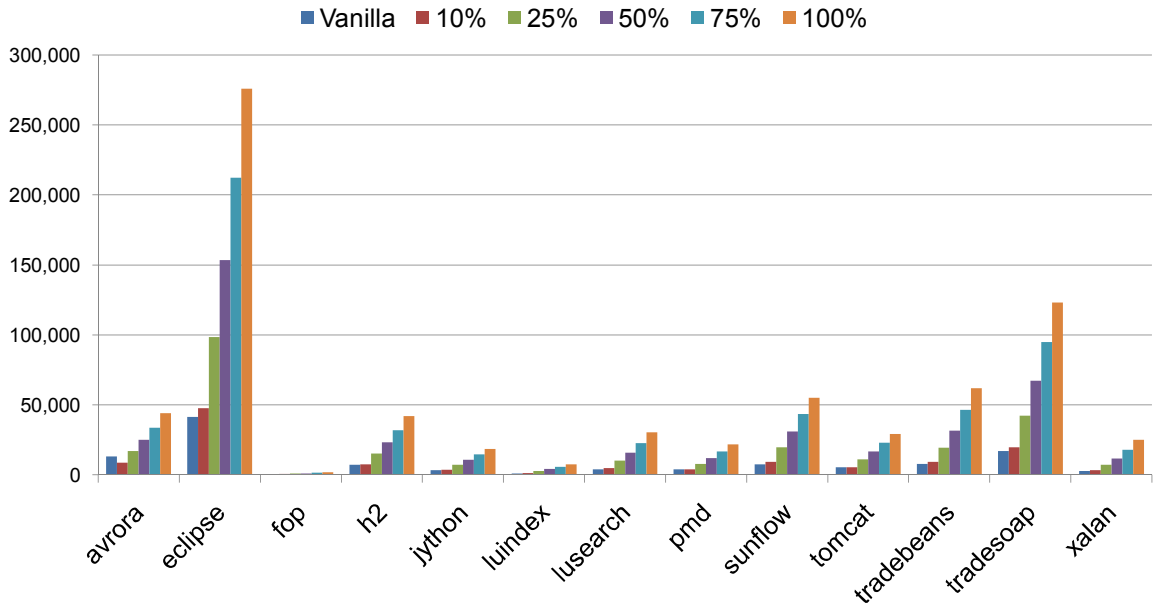

Figure 3.15: Execution time of running instrumented DaCapo benchmarks on MTF.

Table 3.6 shows the number of marker invocations captured by MTF while running the instrumented DaCapo benchmarks.

As shown in Figure 3.15, MTF's overhead is acceptable with small sample sizes, e.g. 10% and 25%. This confirms the intuition that the base overhead is low when the target program has little or no marker instrumentation. However, as the sample size grows, the overheads are increasingly larger. The largest overhead, i.e., 8 times slowdown, occurs when running the fully-instrumented `xalan` benchmark. Larger samples reveal the overhead of frequent context switches caused by the excessive invocations of marker handlers as shown in Table 3.6.

It is to note that the bold numbers in 3.5 of running 10% instrumented `avorora` and `pmd` are smaller than those of the vanilla runs. This does not indicate MTF runs faster than the regular VM does, but is due to intricate VM reactions with the presence of MTF, which might slightly change some VM behaviors, e.g., JIT and cache.

3.5 Conclusions

In this chapter, we present the design and implementation of a flexible dynamic tracing framework, i.e., MTF, for VM-based languages. MTF provides the analysis clients with the ability to define new event types based on program locations represented by markers. Furthermore, rich semantics can be attached to each class of markers by means of a marker descriptor, which can be used to encode analysis-specific data related to the marker. The framework allows flexible handling of each marker invocation that can also be shared by multiple clients. To demonstrate the feasibility and effectiveness of the framework, we have implemented MTF for Java in SUN HotSpot JVM by applying only a small change-set. We also evaluate the performance of the framework by instrumenting and analyzing the DaCapo benchmarks. It is shown that MTF offers great flexibility for developing program analysis clients with low overhead in most applications.

Chapter 4

Adaptive Runtime Typestate Analysis

Modern software engineering practices encourage reusing existing software libraries, e.g., Java Runtime Library, .NET Base Class Library, and C++ Standard Template Library (STL), in implementing new software systems. Most such reusable components have specified interfacing restrictions that must be followed by the developers for implementing well-behaved programs. An example rule regarding the usage of the `Iterator` class in Java specifies that an `Iterator` object has be to queried for availability `hasNext` before advanced (`next`). The violation of such rule may lead to a runtime exception to be thrown, which, if uncaught, may cause the application to crash. Though most such rules are well documented, application developers may not always adhere to these rules. Thus, programming errors due to non-conforming API usages still occur quite frequently in software development.

A *typestate property* describe the set of valid operations that can be performed on an object, depending on the object's typestate [65]. Thus, typestate properties are suitable for representing API constraints. Typestate analysis is an effective technique

for checking whether a program violates given typestate properties. A typestate analysis can be static [11, 31, 16], dynamic [22, 17, 27, 9], or hybrid [18, 28].

In this chapter, we present a *Finite State Automaton* (FSA) based dynamic typestate analysis client that leverages the power and utility of MTF. Furthermore, we apply the *Adaptive Online Program Analysis* (AOPA) [27] optimization to the client to demonstrate the benefits of developing program analyses inside the JVM. We use the instrumentation utility to incorporate typestate properties into the programs to be checked. Our preliminary evaluation of running standard Java benchmarks on top of our JVM shows that the implementation can find actual violations with acceptable runtime overhead.

We structure the remainder of the chapter as follows. In section 4.1, we present the concepts related to FSA-based typestate analysis and AOPA, followed by a description of the MTF-based analysis client. We describe the implementation of the client in section 4.2. Section 4.3 reports the results of our preliminary experiments to show the performance. We conclude the chapter in section 4.4.

4.1 Runtime Typestate Analysis Client

In object oriented programming languages, the type of data objects specifies the set of operations allowed to be executed on them as the receivers. Whereas, typestate determines the subset of these operations that are permitted in a certain context [65]. For example, a `File` can be only read or written after it has been opened; the operating system resource associated with each graphic user interface (GUI) widget must be eventually released after use. Such requirements are commonly seen in software documentation and can be naturally represented as typestate properties, which can be checked for improving software quality.

```
public class File {
  public void open(String name);
  public void close();
  public char read();
  public void write(char c);
  public boolean eof();
}
```

Figure 4.1: The File API.

Several formalisms, e.g., finite-state automaton (FSA) [27, 9], *linear temporal logic* [15], and history-based languages [17], have been used in the past studies. In this work, we used FSA as the underlying formalism to represent typestate properties because: 1) FSA naturally models the dynamic execution stages of a program; and 2) FSA can be easily constructed from *regular expressions* (REGEX) and most developers can conveniently express typestate properties in REGEX. For example, the `File` API shown in Figure 4.1 can be described by the following regular expression:

```
(open (read|eof|write)* close)*
```

Modeling. In our FSA-based analysis, we map each typestate in the property by a state in the FSA. Moreover, we use two special types of states, *source* and *sink*, to represent the the initial state before program executes and the error states, respectively. The switch between two states are represented as a deterministic transition labeled by the event that triggers the transition. Figure 4.2 shows the FSA for the `File` property.

Formally, a typestate property p can be represented it by a FSA — $(Q, \Sigma, \delta, q_0, F)$, where: 1) Q is the set of states; Σ is the set of markers (alphabet); δ is the transition function, i.e., $y = \delta(x)$ $(x, y \in \Sigma)$; q_0 is the source and F is the set of sinks. For the `File` example shown in Figure 4.2, we have: 1) $Q = \{s_1, s_2, s'\}$; 2) $\Sigma = \{$`open`, `read`,

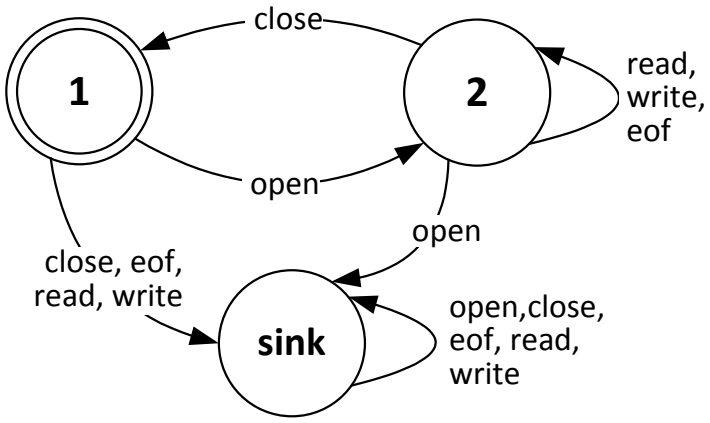

Figure 4.2: The FSA for the File property

`write`, `eof`, `close`}; 3) $q_0 = s_1$; and 4) $F = s'$.

Tracing. Our typestate analysis client needs to continuously trace program flows and record program states to drive the internal FSA for online typestate property checking. This mechanism is supported by MTF. Specifically, we can trace each state in the FSA by placing a marker around the corresponding code region. For the `File` example, we can insert markers at the entries of `open`, `read`, `write`, `eof`, and `exit`, respectively. When a program using `File` is executed, MTF can recognize the markers and raise events to notify the analysis client. The analysis clients can incorporate typestate properties into the target classes using MTF using the layout shown in Table 4.1, where: 1) each method is represented by a transition with a single letter symbol, e.g., **o** for `open`; 2) marker **0** is a special marker to hold the regular expression of the File API property; and 3) each predicate string is formatted as: "`<method-name>,<method-signature> | symbol`".

Checking. The typestate analysis client defines handlers for monitoring the marker events, based on which the FSA is maintained. In addition, each object is associated

Marker ID	Predicate String	Description
0	`(o(r|e|w)*c)*`	specifying the regex of the property
1	`open,(Ljava/lang/String;)V|o`	descriptor for the `open` method.
2	`read,()C|r`	descriptor for the `read` method.
3	`write,(C)V|w`	descriptor for the `write` method.
4	`eof,()Z|e`	descriptor for the `eof` method.
5	`close,()V|c`	descriptor for the `close` method.

Table 4.1: Marker descriptors for the File API property.

with a data structure that records its current typestate information, which is updated by the analysis client at every transition. Every marker event signifies a transition in the FSA according to the property. The analysis verifies the legality of the transition against the property before actually performing the transition and further update the per-object data structure.

Checking the final state of a typestate property for each monitored object requires support from the garbage collector because programs do not explicitly deallocate objects but GC does. Thus, only GC has object death information. To check whether an object dies at one of the final states, we can iterate over the list of dead objects and verify its current state at the end of each collection. Final state checking is critical for many analysis, e.g., resource leak [43, 9] and data structure consistency [45, 26].

Adaptive Online Program Analysis. Because most online typestate analyses incur high overhead and slows down executions by orders of magnitude, recent researches have proposed various optimization techniques [22, 9, 27, 16]. AOPA is one of effective techniques and is based on the observation that at any point during a typestate analysis, only a subset of all transitions can change the program state, i.e., out-going transitions. The other transitions are called *self-loop* transitions where source and destination is the same state. Therefore, events that are symbols to the

State	Self symbols	Out-going symbols
1	{}	Σ
2	{`read`, `write`, `eof`}	{`open`, `close`}
sink	Σ	{}

Table 4.2: Self-loop and out-going symbols in the `File` API [27].

self-loop transitions can be safely ignored to reduce event frequency, leading to a significant reduction in monitoring overhead. Moreover, the set of ignoring symbols are dynamically updated as programs makes out-going transitions. Typestate analysis with AOPA produces the same result as the non-adaptive version but executes more efficiently.

For example, a program has opened a read-only `File` (Figure 4.2), which then stays at state 2 regardless of any subsequently reads, writes, or queries (`eof`), until the eventual `close` is issued. Thus, we can safely apply the AOPA on this `File` object by disabling the monitoring on {`read`, `write`, `eof`} because they are events of self-loop transitions. Since out-going events {`open`, `close`} are being monitored, the analysis can still detect API violations and update the set of self-loop events as shown in Table 4.1. Ideally, there should be exactly two events, i.e. open and close, with all the `read` and `eof` events ignored.

4.2 Implementation

The original implementation of AOPA is called "Sofya" and it is based on the *Java Debugger Interface* (JDI) [67] for intercepting method entry/exit events by setting breakpoints. In this work, we substitute JDI with MTF for the same purpose but with much lower overhead because all executions are stalled when Sofya re-instrument the target classes for adaptive monitoring. Sofya suffers worst-case overhead when the

execution involves mostly out-going transitions.

In this section, we present the implementation details of FSA-based typestate analysis client using MTF.

4.2.1 Finite State Automaton

Our typestate analysis client incorporates `Libfa` [25], an FSA library implemented in C, for the FSA data structures, such as states, transitions, and common operations. The most relevant operations for our analysis are: 1) regular expression *parsing* for FSA construction; and 2) *minimization* for converting NFA to DFA to reduce unnecessary states and transitions. Thus, developers can easily specify typestate properties using regular expressions as inputs to the analysis client, which makes the client generic for any typestate properties. In addition, the determinism resulted from minimization speeds up property checking and eases the analysis code.

To save space, we only keep one instance of the FSA in memory per typestate property. Each Java object holds and refers to only the current state structure, i.e., the states and transitions of FSAs are completely shared. Moreover, multiple Java classes with the same typestate property are checked by a single FSA. For example, all iterators in the program are to be checked by the typestate property `hasNext`, our JVM would cache the regular expression and uses the same FSA instance for all iterator objects. With the two optimizations, our Libfa-based representation has negligable memory footprint.

4.2.2 Per-object Storage

Our typestate analysis performs verification on a per-object basis. Thus, we need to individually monitor and update the state of each target object. There are two

approaches, centralized and distributed depending on the place where the typestate information is stored. In the centralized theme, a global hash table is maintained and indexed by object identifiers. The distributed approach stores information in the object header. Although the per-object storage solution has larger memory footprint due to the added fields, it saves the overhead in looking up and update the hash table, which can be significant when a large number of objects are being monitored.

In this work, we choose to implement the per-object storage solution for efficient retrieval and update of typestate information. Specifically, we can extend the object header as represented by the `instanceOopDesc` structure by adding an extra pointer field pointing to the typestate structure. This scheme adds a 4 or 8 bytes to each object on 32- and 64-bit platforms, respectively. With a single pointer, we trade extra pointer-dereferences for space when we need to access the fields inside the typestate structure.

4.2.3 Adaptive Online Program Analysis

To dynamically switch method instrumentation based on the typestate, `Sofya` utilizes JDI to perform a *redefinition* and *reloading* on the target class at runtime, which suffers high overhead. In this work, we leverage the adaptive marker invocation mechanism as supported by MTF to implement AOPA. Specifically, our typestate client, at each transition, sets the bits corresponding to the markers with out-going symbols whereas clearing those for markers with self-loop symbols. Thus, the instrumentation of methods marked by self-loop symbols are automatically skipped during the subsequent calls. Following the typestate property and FSA transitions, such methods can resume their instrumentation when the symbols they carry become out-going symbols in the future.

4.2.4 Object Death Event Handling

For certain typestate properties, it is essential to verify whether objects die at the final accept state. However, Java uses automatic memory management based on garbage collections. Thus, we need to collaborate with the GC subsystem to capture object death events.

HotSpot by default uses generational garbage collection with a *semi-space* for the young generation and a parallel *mark-compact* collector for the old generation. Thus, the space of the dead objects are simply overwritten by copying the live ones over when reclaimed. Consequently, HotSpot does not trace object death event. Fortunately, HotSpot supports a mechanism called `JNI handle`, a managed pointer that is transparent to the referrer when the referenced object is relocated in memory by GC. When a Java object is referenced by a JNI handle, GC cannot reclaim the object. Unlike regular JNI handles, weak handles do not withhold GC from reclaiming the referenced objects, similar to a *weak reference*. We leverage the heuristic that the weak handles of dead objects resolve to `NULL` pointers.

Since tracking dead objects requires special support from HotSpot, an *object death event* context provider (ODE) is implemented to provide such feature. To track dead objects after each GC, ODE keeps a linked-list of `typestateHandle` structures, i.e., pairs of per-object typestate and weak JNI handle for all monitored objects. ODE registers the GC-end event where it can check the liveness of the objects by repeatedly resolving their JNI handles. For each dead object, it notifies the handlers that register the object-death event with the object's `typestateHandle` structure. Our typestate analysis client registers the object-death event provided by ODE. Thus, it can verify the eventual typestate where each object dies at.

To minimize the overhead from invoking handler, we bundle the `typestateHandle`

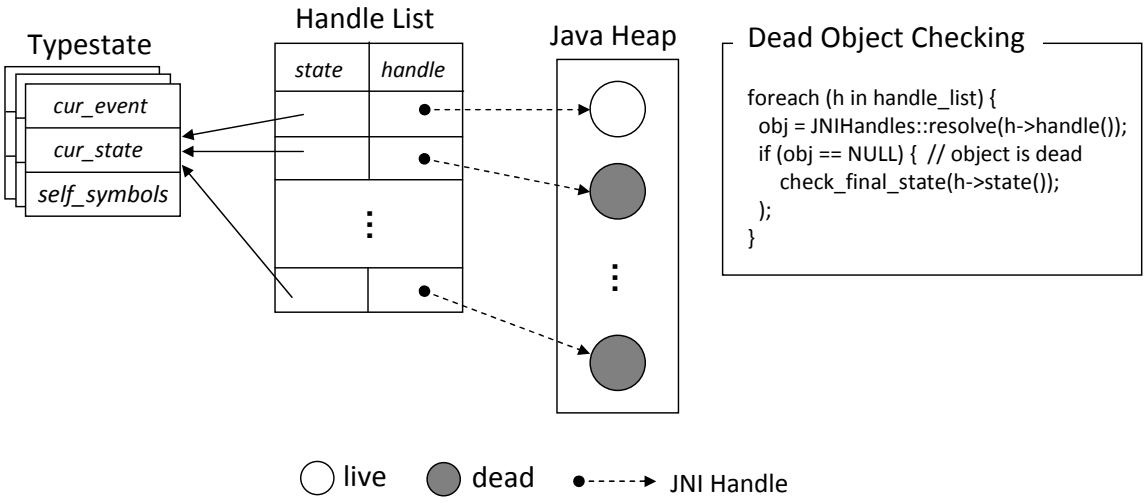

Figure 4.3: Dead object typestate checking.

structures of all dead objects on a linked-list and pass the list to the handlers as a batch. To further reduce the overhead of creating a different list, we simply slice out the live objects onto a new list because we observe much more deaths than lives in profile runs. Thus, the `typestateHandle` of dead objects are left in-place on the original list, which can be used as the bundle. In the end, the list with the live objects becomes the new handle list for the next pass. With such optimization, a single GC execution triggers only one method call per registered handler. Figure 4.3 illustrates the mechanism of ODE and our optimization.

4.3 Evaluation

In this section, we present the evaluation of our adaptive FSA-based dynamic typestate analysis powered by MTF with a set of experiments.

4.3.1 File API

We show the effectiveness of both the analysis and optimization by verifying the File API on a *micro-benchmark* suite as shown in Table 4.3. The suite contains conforming programs and non-conforming ones with manually injected violations to show the effect of the optimization and functionality of the typestate analysis, respectively.

Benchmark	Description
Main	Open two files for reading and writting (conforming).
NoEof	Open a file for reading but never call `eof()`.
NoOpen	Read a file that is not opened.
NoClose	Read a file but do not close it in the end.

Table 4.3: Micro-benchmarks to test the File-API property.

Except *Main*, other benchmarks all violate the File API property in certain ways. *NoClose* is a special benchmark because it tests the ability of the framework to capture the object death event, which is necessary for verifying the final state where each target object dies. For example, our analysis should be able to detect in *NoClose* that the file is kept unclosed when the program ends.

In the experiment, the typestate client successfully found and reported all the violations. For example, it reported the violation

```
Found invalid transition ^ -> r (tests.file.File) <tid=14068>
```

while executing the *NoOpen* benchmark which reads (`r`) the file (`tests.file.File`) that is not opened (`^`) by thread `14068`. For dead object violations, the client reports

```
Dead object (0x01ebcb48) violation @ {r} state
```

for the *NoClose* program, indicating that the object at memory address `0x01ebcb48` dies at the *read* state and violates the property requiring the final state to be *close*.

For *Main*, we record the numbers of marker invocations before and after the application of the AOPA to show its effectiveness. Because the benchmark is short-running, we do not make comparisons based on the run time, but on the reduction of marker invocations. We executed *Main* five times and reported the best, the worst, and the geometric mean in Table 4.4 and 4.5.

Non-adaptive			Adaptive		
Max	Min	Geomean	Max	Min	Geomean
6,000,008	6,000,008	6,000,008	31,224	29,418	30,439

Table 4.4: Marker invocations of the File-API benchmark.

Non-adaptive (msec)			Adaptive (msec)		
Max	Min	Geomean	Max	Min	Geomean
378	444	401.0	22	22	22

Table 4.5: Execution time of the File-API benchmark.

4.3.2 DaCapo

We check the usages of iterator objects in the DaCapo suite to further show the performance and effectiveness of our client. The intended usage of `java.util.Iterator` classes requires that `hasNext` must precede each call to `next`, which is commonly referred to as the *HasNext* property. In this experiment, we simplify HasNext into a new property, called *HasNextOnce*, i.e., `hasNext` must be called at least once prior to any subsequent `next`. We show the state machines of both HasNextOnce and HasNext in Figure 4.4. As shown in the FSA, HasNextOnce has the adaptive structure because we can eliminate the instrumentation as soon as we see a `hasNext`. Thus, we expect adaptive typestate analysis to work well for `hasNext`. On the other hand, HasNext

represents the worst-case scenario for AOPA because each time `next` is called, the monitoring of `hasNext` must be restored, thus no instrumentation can ever be disabled.

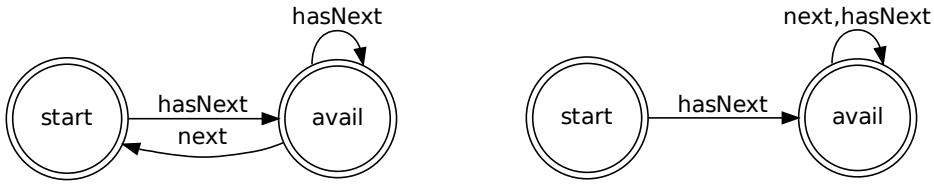

Figure 4.4: Finite-state automatons for HasNext (left) and HasNextOnce (right).

In our preliminary studies, we observe that the `bloat` program in the October 2006 release of DaCapo (`2006-10-MR2`) has the most utilization of iterator class. Thus, we include `bloat` in this experiment along with the programs in the newer DaCapo release (`2009-12`). Moreover, `tomcat`, `tradebeans`, `tradesoap`, and `xalan` do not utilize iterators in their execution, thus are excluded in this experiment.

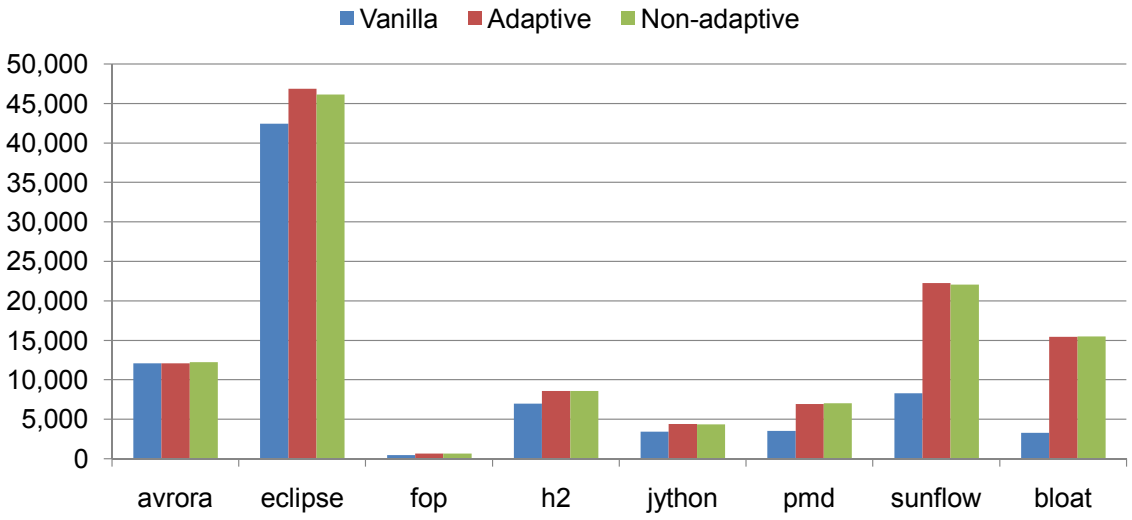

Figure 4.5: Execution time of DaCapo with HasNext property.

In Figure 4.5 and 4.6, we report the execution time of each DaCapo benchmark with HasNext and HasNextOnce property, respectively. Beside marker invocation counts, Table 4.3.2 and 4.3.2 also show the reductions of execution time and invocations achieved by the AOPA.

		Adaptive		Non-adaptive		Reduction		
Benchmark	Vanilla	Time (ms)	Event	Time (ms)	Event	Time (%)	Event (%)	Monitor
avrora	12,066.60	12,095.60	26	12,230.80	26	1.11%	0.00%	2
eclipse	42,476.00	46,894.00	99,740	46,158.00	140,768	-1.59%	29.15%	3,613
fop	445.60	633.00	176,155	623.40	181,353	-1.54%	2.87%	15,751
h2	6,962.10	8,565.30	25,334,946	8,560.40	25,334,340	-0.06%	0.00%	6,519,800
jython	3,415.00	4,366.40	31,552	4,355.20	307,590	-0.26%	89.74%	24,045
pmd	3,519.00	6,917.20	1,562,249	6,996.20	1,868,551	1.13%	16.39%	238,781
sunflow	8,287.80	22,254.50	3,931,300	22,032.60	3,931,595	-1.01%	0.01%	100,956
bloat	3,277.20	15,418.80	148,490,519	15,496.00	149,149,959	0.50%	0.44%	149,149,959

Table 4.6: Execution time and marker invocations of HasNext property.

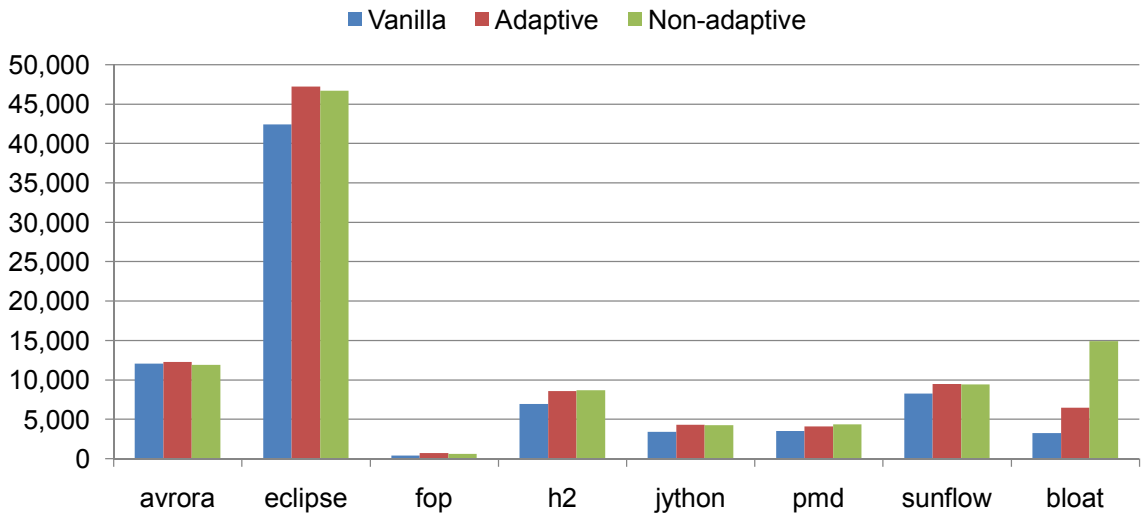

Figure 4.6: Execution time of DaCapo with HasNextOnce property.

		Adaptive		Non-adaptive		Reduction	
Benchmark	Vanilla	Time (ms)	Count	Time (ms)	Count	Time (%)	Count (%)
avrora	12,066.6	12,308.0	2	11,937.8	26	-3.10%	92.31%
eclipse	42,476.0	47,273.0	3,613	46,714.0	140,774	-1.20%	97.43%
fop	445.6	723.2	16,205	635.6	181,353	-13.78%	91.06%
h2	6,962.1	8,596.7	6,519,560	8,692.3	25,330,249	1.10%	74.26%
jython	3,415.0	4,353.8	26,154	4,255.0	307,590	-2.32%	91.50%
pmd	3,519.0	4,102.2	566,214	4,374.6	1,868,874	6.23%	69.70%
sunflow	8,287.8	9,517.4	1,365,234	9,477.2	3,922,933	-0.42%	65.20%
bloat	3,277.2	6,503.2	941,270	14,920.6	147,604,481	56.41%	99.36%

Table 4.7: Execution time and marker invocations of HasNextOnce property.

Note that while we can achieve over 90% reductions in the number of events, they have not directly translate to reductions in time as indicated by negative reductions. Such results are due to that the overhead of applying the optimization outweighs that of processing markers, when the number of marker invocations are not large enough. For example, even without the optimization, `avrora` only has 26 invocations such that the optimization becomes an overkill. Nevertheless, the measurements of marker invocations in both tables clearly prove the effectiveness of the optimization in reducing unnecessary marker events. The most representative case is the HasNextOnce property on `bloat`, which has the heaviest usage of iterators among all benchmark

programs, with 60.78% and 99.36% reduction in time and events.

In summary, we can conclude that when the number of marker events is sufficiently large, e.g., over 1,000,000, it is worthwhile to apply the AOPA since it can significantly reduces the instrumentation overhead. Otherwise, such optimization cannot yield better performance because the inherent overhead might outweigh the gain from the event reduction.

4.4 Conclusions

In this chapter, we present the design and implementation of our dynamic typestate analysis client on top of MTF. The client uses FSA for typestate property checking with AOPA optimization, which is implemented by the method preamble dispatching technique. To support verification of the final typestate where object dies, we exploit the JNI handle heuristic and capture object death event at the end of each garbage collection. Experiment results show that our typestate analysis client is able to detect property violations and incurs acceptable overhead. Our implementation allows for incorporating complementary techniques, e.g., sampling and static analysis, to further reduce the overhead. Such enhancements remain as future work.

Chapter 5

Selective Probabilistic Calling Context

Dynamic calling context is the set of active method invocations on the call stack during program execution. Calling context is an essential tool for developers to understand as well as troubleshoot programs because it reveals precisely the execution path down to the current location. When debugging information is readily available, each activation can be mapped to the exact lines of the source code. For example, when a memory error happens, the developer can locate the faulting code by examining the calling context as a stack trace at the error location using a source-level debugger, e.g. *The GNU Project Debugger* (GDB). It is very hard to efficiently track the dynamic calling context because of the huge number of method invocations in most applications. To reduce the overhead in computing dynamical calling context, past studies have proposed many optimizations, e.g., *Probabilistic Calling Context* (PCC) [20], *Inferred Call Path Profiling* [52], and *Precise Calling Context Encoding* [66].

In this work, we implement an enhanced PCC, called *Selective Probabilistic Calling Context* (SPCC), via an analysis client to the MTF framework with the purpose of

computing calling context for a selected subset of all methods, whereas standard PCC computes calling context values for all application methods. The focus is to demonstrate the applicability of MTF in improving existing runtime analyses like PCC. This chapter first reviews the concept of PCC and show how we extend PCC with MTF into SPCC. Then, we describe our experiences in porting the original PCC onto HotSpot JVM as well as the implementation of the SPCC client. We also show the experiment results of running SPCC with the DaCapo benchmarks.

5.1 Probabilistic Calling Context

PCC uses an integer value V to represent and continuously track the current calling context. PCC strives to compute a statistically unique number for each context. Thus, a PCC-based context profiling typically involves two runs: 1) a *training* run aiming at collecting PCC values at interesting locations; and 2) a *production* run for recording calling contexts for the collected PCC values.

Computation. PCC adds instrumentation that computes V at each call site by applying a function f as follows [20]:

```
method() {
        int temp = V;      // PCC: load PCC value
        ...
        V = f(temp, cs_1); // PCC: compute new value
  cs_1: calleeA(...);      // call site 1
        ...
        V = f(temp, cs_2); // PCC: compute new value
  cs_2: calleeB(...);      // call site 2
        ...
}
```

The following function f is used to compute the PCC values:

$$f(V, cs) = 3 \times V + h(cs)$$

where V is the value of the current context and h is a hash function giving the random number for cs. It has been shown in [20] that function f has the essential properties: 1) computes PCC values with acceptably small level of conflicts; 2) deterministic (same calling context always computes the same value); and 3) efficient. Moreover, f is non-commutative it is mandatory for f to compute distinct values for same methods but different orders, to differentiate such contexts. h is computed for each call site by hashing the method and line number. The hash value of each method can be computed statically at class loading time by hashing the name and signature as the method such that h can be easily computed subsequently.

Query. During a profile run, we can collect and record the PCC values of interesting program locations during execution. In the tracing run, the VM can recognize the exact locations according to the recorded values and track the calling contexts and presents them with precise information in stack traces with line numbers.

5.2 Selective Probabilistic Calling Context

We extend PCC with MTF to support selectively computing calling contexts for a subset of all methods. This approach is called *Selective Probabilistic Calling Context* (SPCC). SPCC inherits all the advantages of standard PCC. Additionally, SPCC can effectively lower the overhead of PCC since fewer call sites would result in less computation. SPCC provides context sensitivity for other peer clients within MTF by allowing them to control the set of methods participating in the calling context computations. Essentially, SPCC introduces fine-grained filtering mechanisms, both offline and online, over the original PCC.

SPCC Marker. For method filtering, SPCC analysis client introduces a special kind of marker, i.e., *SPCC marker*, which, when present, indicates the current method requires PCC computation. One usage of SPCC marker does add marker bytecodes into the application but only for method tagging purpose. A second usage is based on marker bytecodes which triggers a custom routine for adaptive control of PCC computation.

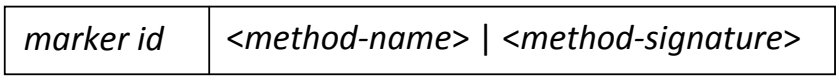

Figure 5.1: Marker descriptor format for SPCC.

The descriptor of each SPCC marker consists of a common *fake* marker id and a distinct predicate string in the format of a pair of method name and signature as shown in Figure 5.1.

Compile-time Filtering. SPCC client extends the header of each method structure to include a flag, which is set for all instrumented methods. The flags of the uninstrumented methods are cleared. During the training run, SPCC checks the flag before computing the PCC value at each call site. The flag is also consulted at the method entry and exit such that methods with cleared flags do not cache PCC values since they are not interesting to the clients.

The SPCC analysis client registers the `subscribe` and `method_loaded` events of the handler interface for setting flags on the methods in the corresponding classes. Because both events happen only once for each method during class loading, the SPCC client introduces negligible overhead over the standard PCC.

Runtime Filtering. With the assistance of MTF, SPCC also features find-grained runtime control similar to the typestate analysis client. Users can add SPCC marker

bytecodes at arbitrary program locations to raise events at runtime. To receive such events, the SPCC client can register the `markerenter` and `markerexit` events to perform any desired analysis and adjustments. For example, SPCC supports adaptive switching on and off PCC computations for certain methods based on the analysis results by setting and clearing the per-method flag in the marker handler. With such capability, SPCC can further lower the overhead of PCC as it provides a *programmable* runtime filter over the existing methods selected at compile-time.

5.3 Implementation

The original PCC implementation is based on Jikes RVM [5], an open-source *meta-circular* [70] JVM implemented in Java. However, HotSpot is programmed in mixed C++ and platform-specific assembly language. Hence, re-implementing PCC on HotSpot is a non-trivial task even though we have access to source code of both Jikes RVM and PCC.

In this chapter, we describe our implementation of SPCC, i.e., a port of the original PCC and an MTF-based analysis client on HotSpot. We highlight the design decisions as well as extensions we have made to bridge the technical differences between Jikes RVM and HotSpot.

5.3.1 PCC Stack

To maintain the V values throughout method invocations, original PCC instrument the entry and exit of each method with a load and store operation respectively. Loading the V value into a temporary slot on the Java stack saves the current PCC value which can then be restored by the store operation when method exits. Unlike Jikes RVM, HotSpot has a very complex frame layout system, e.g., interpreted, compiled, native,

and runtime method each has a special frame layout. Moreover, HotSpot has fourteen entries for different types of methods for optimizations. For such technical reasons, we choose not to modify the frame layout as original PCC does, but use a dedicated buffer as the PCC stack. Such PCC stack is thread-local such that it can support multi-threaded applications on multi-processor machines. Each frame on the stack has only one entry — the PCC value. Thus, at any time during execution, the number of PCC stack frames equals the number of active methods on the calling context. Since each PCC takes up 4-bytes in memory, a 256 KB stack can accommodate 65,536 method activations. Except for extremely deep recursions or massive parallel programs, this solution should be able to support most real world applications with an acceptable memory footprint.

Two modifications are necessary for maintaining the PCC stack in the byte-code interpreter and JIT compiler. For the interpreter, we extend the existing method entry and exit hooks: `notify_method_entry` and `notify_method_exit` in the `InterpreterMacroAssembler` class by adding assembly code for saving and restoring PCC values. For compiled methods, we extend the JIT compiler by constructing IR nodes that perform the stack update in the `do_method_entry` and `do_method_exit` code in the `Parse` class, which is the parser of the C2 compiler. The following pseudo code presents a high level view of the stack update operations we have added to both the interpreter (assembly code) and JIT compiler (IR nodes).

```
method_entry() {
 pcc = thread.pcc_value();
 *(thread.sp) = pcc;
 thread.sp --;
}
```

```
method_exit() {
 thread.sp ++;
 saved_pcc = *(thread.sp);
 thread.set_pcc_value(save_pcc);
}
```

5.3.2 Computing

PCC values are computed at call sites, i.e., program locations where methods are invoked, for both interpreted and compiled methods. Similar to the method entry and exit codes, we incorporate the PCC computation code into the interpreter and JIT compiler.

JVM specification [47] defines four bytecodes for invoking methods, i.e., `invokeinterface`, `invokespecial`, `invokestatic`, and `invokevirtual`. Thus, the occurrences of these bytecodes indicate call sites where a new PCC value needs to be computed. Thus, we extend the code templates of these bytecodes in the `TemplateTable` class by adding the PCC computation code. Likewise, we build IR nodes performing the same computation in the the `Parse::doCall()` method, where IR nodes for compiled method invocations are generated. the current As described in section 5.1, each call site is represented by a unique hash code computed by function h. Same as the original PCC, h consists of two parts: 1) hashing of method; and 2) hashing of the location of the invoke instruction. h adds the two hash values to get the hash code for each call site.

HotSpot represents each string symbol by a `symbolOopDesc`. To save space, HotSpot stores all symbols in a `symbolTable` by hashing the string content using the hash algorithm developed by Kernighan and Ritchie [41]. Thus, we reuse existing hash codes of the class name, method name, and method signature by simply adding them up.

Hashing the invoke instruction is done by using its line number to query a fixed random number table, which is created at VM startup. Unlike the original PCC, we do not use line number to represent the location of the invoke bytecode because line number table cannot be accessed efficiently by the interpreter or JIT compiler. Thus, we instead use the *bytecode index* (BCI), i.e., the offset from the first instruction of the

method. Interpreter derives the BCI by minus the readily available bytecode pointer (BCP) by the method base. JIT already has access to the BCI by examining the code stream being compiled. We use a fixed length random number table with 2^{16} entries. Thus, the BCI is rounded by masking off the upper bits and used to table lookup: `random_table[bci & 0xffff]`.

5.3.3 Query

The original PCC supports querying for a pre-defined set of methods, e.g., system calls and library calls. With our MTF-based SPCC implementation, we can support querying PCC values at arbitrary program locations during execution by tracing SPCC marker bytecodes. We have developed a new client in the instrumentation utility to add SPCC markers to programs. The SPCC client handles these markers at runtime such that it can records the PCC values for the interesting events based on the results of any analyses.

We provide a command line argument, i.e., `TracePCCs` that can accept a list of PCC integers to be traced in the production run. These PCC values are stored in a hash table for efficient lookup by the PCC computation routine in both the interpreter and JIT compiler. For each matched PCC value, the calling context is retrieved by walking the call stack. Moreover, we convert BCIs into line numbers for better presentation. All recorded contexts are stored in a linked-list, which is dumped at the end of execution. With method inlining, a single physical stack frame might correspond to several inlined methods. Thus, HotSpot uses `vframe` as virtual stack frames to represent source-level activations. Walking `vframe` stack can be done simply by using the `vframeStream` class, which provides an easy-to-use iterator-like interface. We repeatedly build up a `stringStream` and stores the content into the corresponding

entry in the hash table.

5.4 Conclusions

In this chapter, we have described how MTF can be utilized to improve existing dynamic analyses, e.g., Probabilistic Calling Context (PCC). We also show the design and implementation of the Selective PCC (SPCC) analysis client, consisting of a PCC port on HotSpot and a companion analysis client based on MTF. SPCC enhances the original PCC by providing two filtering mechanisms, i.e., compile-time and runtime filtering. Such mechanisms help to reduce runtime overhead and offer finer-grained calling context computation. SPCC also introduces a new way for querying and recording PCC values at locations defined completely by developers using SPCC marker bytecodes.

The original PCC is implemented in Java on Jikes RVM, which has a drastically different design as HotSpot. Therefore, we describe our experience in porting PCC onto HotSpot while presenting our implementation in the final section.

Chapter 6

Related Work

This chapter describes the existing techniques that are most relevant to the marker tracing framework. To conclude the chapter, we summarize the techniques along with MTF based on a set of criteria.

6.1 Java Virtual Machine Tool Interface

HotSpot JVM implements an infrastructure called: *Java Virtual Machine Tool Interface* (JVMTI) [2], a comprehensive programming interface used by developer and monitoring tools. Numerous prior works are based on JVMTI, e.g., dynamic program analysis [12, 57, 64], mixed-environment debugging [46], performance monitoring [61, 50], and fault injection [38, 63].

JVMTI allows a user-supplied *agent* (a client of JVMTI) to access internal VM states and control program executions. Agents can receive event notifications when the registered events are triggered. Moreover, JVMTI passes arguments to the callback functions to provide additional information about the event. Example JVMTI events include but not limited to `VMStart`, `ClassLoad`, `FieldAccess`, `MethodEntry` and

`MethodExit`. Through JVMTI functions, agents can query various program states, e.g., stack traces, thread state, local variables, object monitors, and loaded classes. JVMTI also enables agents to alter program executions, e.g. suspending threads, setting breakpoints, and popping stack frames.

As a fixed interface, JVMTI limits the clients to the existing event types and accessing functions. To address this issue, JVMTI supports *bytecode instrumentation* such that users can insert extra bytecodes in the classes to capture the unavailable events. This can be done at compile time and load time, or during program execution with `RedefineClasses`. Since bytecode manipulation is not directly supported by JVMTI, we have to use 3rd-party tools, e.g., ASM or BCEL, to transform bytecodes before passing to JVMTI. The ability of JVMTI to allow such types of instrumentation enables it to implement certain portions of our MTF framework. However, we argue such solution is not as flexible or efficient as the MTF framework. First, JVMTI agents do not have access to all VM resources at disposal as MTF clients do. Second, `RedefineClasses` is an expensive operation involving class re-loading and re-parsing, in addition to the overhead of online bytecode transformation. Whereas, with techniques such as AOPA, MTF supports efficient program instrumentation switching with much lower overhead. Lastly, it is non-trivial and not as integrated to manually replicate the features of MTF as JVMTI agents.

Nevertheless, it is promising and beneficial to integrate MTF into JVMTI such that MTF clients can leverage existing functions and events provided by JVMTI. In addition, new facilities that are unavailable by JVMTI can be added along with MTF clients into the VM as extensions, which in turn enhances the JVMTI infrastructure.

6.2 Dynamic Tracing

Dynamic Tracing Framework (DTrace) is a component in Solaris 10 operating system. DTrace is a powerful infrastructure for administrators and developers to explore arbitrary behaviors of the operating system and user programs with very low overhead. DTrace supports collecting performance metrics in the production environment by dynamically modifying the operating system kernel and user processes at locations of interest, i.e., *probes*, which are made available by *providers*. By writing programs in the *D* programming language, users can precisely and concisely specify the probes to enable and actions to perform when the probes are hit. DTrace allows probe filtering via *predicates*, which are evaluated at runtime. All instrumentation in DTrace is completely dynamic, i.e., probes are enabled only when they are used and no instrumentation is present for inactive probes. Thus, the rest of programs outside probes run at full-speed.

The Java Platform, Standard Edition 6 (Java SE 6) introduces DTrace support in the HotSpot JVM with two DTrace providers: `hotspot` and `hotspot_jni`, which are built as JVMTI agents. The `hotspot` provider supports probes within various subsystems in HotSpot, e.g., VM lifecycle, garbage collection, class loading, JIT compilation, object allocation, and method entry/exit. Similar to the JVMTI, `hotspot` and `hotspot_jni` contains a fixed set of probes and do not support bytecode instrumentation, which is essential for extending the existing event interface. Whereas, MTF is designed specifically to allow users to define custom events based on arbitrary program locations of interest and attach callback handlers, which can access all VM services and potentially have complex logics. Moreover, DTrace is platform-dependent since it relies on supports inside the OS kernel. MTF, on the other hand, can be supported on all paltforms regardless and does not rely on any OS services. Nonetheless, we

speculate that D scripts, when extended, can be used for dynamic marker specification for MTF, as an alternative for compile-time constant pool based solution.

6.3 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) has also been used for profiling [58, 7] and dynamic program analyses [72, 6]. In this section, we consider AspectJ [42], the most widely used AOP implementation for Java. AspectJ supports systematic program instrumentation by providing language constructs for describing instrumentation points (*join points*) and adding custom actions (*advices*). An advice can be attached to either a single join point or a set of them with *point cuts*. Example join points supported by AspectJ include: method calls and executions, types of sender and receiver objects, exception handlers, and control flows. Advices are written in standard Java language and have access all Java libraries. In addition, aspectJ supports advice weaving at both compile-time and load-time with the AspectJ compiler (`ajc`) and *weaving class loaders* respectively.

Unlike JVMTI and DTrace, AspectJ-based instrumentation does not require special support from the JVM or the OS kernel, as the target program is modified directly. Thus, AspectJ-based instrumentations are easier to develop and more portable than JVMTI agents and DTrace scripts. However, this kind of instrumentation is restricted to only user level information, e.g., receiver objects and method arguments. At such level, internal VM information is completely inaccessible to AspectJ. For example, AspectJ-based instrumentation can interact with neither the garbage collectors nor the JIT compilers. Furthermore, the weaved aspects are full-blown Java code that not only bloat the target programs but can introduce noticeable runtime overhead. MTF solves both problems by implementing clients as part of the VM using only single or

two bytecodes, although MTF clients are more difficult to implement. Despite the drawbacks of AspectJ for fine-grained low-level program instrumentation, its language and compiler can potentialy be extended for MTF users to systematically specify markers; such solution can sometimes be more convenient than developing a client on top of our BCI-based instrumentation utility.

6.4 Quality Virtual Machine

Quality Virtual Machine (QVM) is a specialized runtime environment on top of IBM's J9 JVM [9]. The objective of QVM is to provide an infrastructure for detecting software defects that occur in the post-deployment stage in production environment. QVM continuously yet efficiently monitors the execution of the application against user-specified correctness properties, e.g., typestate properties, Java assertions, and heap properties.

To control the overhead, QVM uses a novel *overhead manager* to enforce a user-specified overhead budget. QVM collects as much useful information as possible from the executing program while staying within the specified budget with *object-centric sampling*, which allows sampling at object instance level. Analysis clients receive profile events only from the objects that are marked as tracked as indicated by a bit in the object header. QVM samples the objects based on the allocation sites and uses a short inlined code sequence to check the tracking bit before any QVM callbacks are made. Thus, QVM can adjust the sample size at each allocation site to control the event frequency such that the overhead can stay within budget. For fair sampling, sites with lower allocation frequencies receive larger sampling quanta. QVM supports *emergency shutdown*, i.e., discarding a hot and long lived object to avoid severe performance degradation.

Similar to MTF, QVM supports VM side event filtering by allowing users to specify the methods to be monitored such that the rest of the program runs at full speed. However, QVM instruments programs at method or field level, not at basic block level as MTF does. Thus, it is impossible to trace only a subset of all the statements in a method. Nevertheless, QVM's approach in controlling the monitoring overhead based on fine-grained adaptive property-guided sampling is highly effective. Since MTF is also a VM-level solution, it can also implement such mechanism and achieve similar performance guarantee as QVM. Moreover, the assertion and heap property clients are readily ported to MTF. Lastly, MTF excels QVM by supporting user-defined events which greatly improve the flexibility and versatility for program instrumentation.

6.5 Summary

For different objectives, program instrumentation frameworks have different trade-offs. We have developed the following criteria to evaluate the frameworks introduced in this chapter and highlight our objectives and trade-offs in designing MTF.

Flexibility. A flexible framework needs to suit the development of program anlayses by supporting: 1) fine-grained specification of instrumentation points; 2) user-defined event types; and 3) runtime dynamic update of instrumentation. All frameworks support instrumentation points at method and field level. MTF improves the state-of-the-art by allowing specifying instrumentation points within basic blocks. Though JVMTI supports user-defined events by bytecode instrumentation, such mechanism has high costs and is not as powerful as MTF's approach. All techniques support dynamic instrumentation update to certain extent. However, the mechanisms are either too costly (class redefinition in JVMTI and probe predicates in DTrace), or primitive and

high level (advices in AOP). QVM and MTF address such limitations using dynamic compilation techniques, which are more flexible and have lower overhead.

Observability. To assist the development of program analyses, instrumentation frameworks should provide the analyses with runtime observability, i.e., the access to various runtime information. Both JVMTI and DTrace offer comprehensive sets of routines for accessing such information. As standard APIs, such routines are fixed and cannot meet every need that real world program analyses might have. AOP-based analyses are limited to program level information, thus cannot perform operations that depend on low level access. Being VM-based solutions, clients of QVM and MTF have all information and runtime services at disposal. Consequently, QVM and MTF give the clients most observability, albeit such observability comes with the prices of less abstraction, worse portability, and technical complexity of VM modifications.

Overhead. Both DTrace and QVM are designed with performance as one of the most important rationales since the targets are deployed systems in production environments. On the other hand, JVMTI- and AOP-based analyses typically have higher costs due to their respective mechanisms. In reality, the largest instrumentation overhead can be attributed to the number of instrumentation events. Thus, reducing the event frequency is the most effective optimization. Among all frameworks, only QVM and MTF supports VM side event filtering, which are much more efficienty than client side filtering as JVMTI, DTrace, and AOP-based analyses do. In addition, MTF clients can be even more efficient because the filtering is performed at a finer granularity (basic blocks) than QVM clients (methods).

Chapter 7

Future Work

In this chapter, we describe the main areas of future work related to the Marker Tracing Framework (MTF) and the analysis clients. The first area is to develop new techniques in reducing the runtime monitoring overhead resulted by MTF. The second area is to make improvements over the existing implementation of MTF and the analysis clients. The effort of this area is to make MTF more usable. The last area shows a new analysis client we plan to develop based on MTF to further enhance program observability.

7.1 Runtime Overhead Reduction

As described in the previous chapters, the current design of MTF still can result in significant runtime overhead under highly frequent marker events, especially when the analyses require more sophisticated runtime support, e.g., locking and memory allocation. We observe that the dominating factor of the overhead comes from the frequent state transitions between the application and VM states. During such transitions, several expensive operations maybe performed such that the consistency of

the VM is ensured. For example, HotSpot needs to update thread state and deallocate unused resources before the final transition. Thus, the aggregated transition overhead is proportional to the number of events. To reduce such overhead, we can approach in two directions:

- simplifying the transition operations; and
- reducing the number of runtime marker events.

We have taken the first approach in our implementation by separating the clients into two groups, i.e., simple and complex, and treat them differently at runtime. For simple clients, we do not perform the expensive transition operations. For complex ones, we have to issue the transition operations for proper execution. For simple analysis, this solution works well such that the runtime overhead is acceptable event under heavy event load. However, it has no effect for complex analysis clients, which are more interesting and capable in solving real world problems.

State transition overhead is a fact of life when the analysis clients might intervene with the normal VM execution. Thus, only by lowering the event frequency can reduce the resulting proportional overhead. To this end, we identify and summarize two promising techniques that can help achieve such goal.

Static Analysis. There is a stream of research that attempts to use static analysis techniques to reduce the runtime overhead of dynamic analyses [16, 18, 29] by reducing the number of deployed monitor probes. MTF can benefit from the results of such static analyses and performs more efficient online tracing for certain analysis clients. For example, we can design special markers to guide MTF to skip the events that are provably violation-free for the typestate analysis client. We are optimistic that such hybrid approach is promising to significantly reduced runtime overhead.

Sampling. Past studies have investigated sampling-based techniques to reduce the overhead of dynamic analyses. However, naïve sampling is incomplete due to possible false positives and negatives. Recently, QVM [9] has implemented a highly precise and fine-grained sampling scheme, which supports user-defined overhead threshold and can dynamically adjust the sampling rate accordingly. Its *overhead manager* samples objects to be monitored by their allocation sites. Specifically, QVM samples less objects at the sites that are allocation intensive to keep the accumulated overhead under the threshold. For the sampling to be fair for the sites that allocate object infrequently, QVM dynamically increase their sampling rates to raise the chance.

We are planning to adapt QVM's sophisticated sampling technique and implement it in our solution.

7.2 Implementation Improvements

In the remainder of this section, we describe improvements that can be made to our MTF.

Markers as VM Intrinsics. Currently, markers are implemented as two new Java bytecodes, i.e., `markerenter` and `markerexit`. However, extending the Java instruction set is not the best way to support new semantics in the JVM. First, programs instrumented with marker bytecodes cannot be loaded by other JVMs. Second, such programs cannot be processed by existing bytecode utilities and libraries, e.g., `javap` for classfile printing and `SOOT` for program analysis. Third, supporting new bytecodes requires extensive modifications of the JVM code, though many of them are unnecessary boiler-plate code. Lastly, JVM has no knowledge of how to apply optimizations on such bytecodes i.e., marker bytecodes can pollute JIT compilation.

In the future work, we propose to support markers by native methods implemented as VM *intrinsics*. Example intrinsics include: `Object.getClass`, `Class.isInstance`, and `java.math.sin`. This solution can mitigate the aforementioned drawbacks because all existing libraries, utilities, and JVMs can easily handle methods. Furthermore, the existing optimizations in the JIT compilers are readily available for methods with markers invocations. Lastly, adding a new native method saves the unnecessary modifications in the irrelevant modules, thus leading to more maintainable and portable code.

Constant Pool. MTF stores analysis-specific information in the constant pool area of each classfile because of the convenience and portability. Nonetheless, this solution has several drawbacks. First, such binary format is neither human-readable nor manually editable. It would be desirable allowing easy modification by storing the data in a text-based *satellite* file alongside each classfile. Second, JVM specification [47] requires each string constant in the constant pool must be unique. Thus, ASM eliminates duplicate strings before writing the classfiles. However, marker-related information is not always represented by distinct strings, which can lead to an ill-formed marker specification. Currently, we add trailer data to ensure the uniqueness of the marker-related strings. Currently, we are exploring alternative mediums for storing marker meta-data, e.g., *annotation*.

Native Method. In Java, each *native method* is implemented by the host JVM typically in a different language than Java. The JVM knows all the built-in native methods and handle them differently. Example native methods include: `System.gc()`, `FileInputStream.read()`, and `Object.getClass()`. Since such methods are not implemented in Java, the instrumentation utility cannot add markers inside them.

Alternatively, the markers have be inserted around their call sites. However, due to *reflection* and *polymorphism*, it is very difficult to reliably identify all such call sites. Thus, we cannot trace native methods easily using MTF. One possible solution would be to extend the VM to call back MTF when each native method is executed. As such, at least each entry and exit becomes traceable.

Multi-object Typestate Analysis. Besides the single object properties we have discussed in Chapter 4, multi-object typestate properties have also been used for verifying more complex API usages involving multiple interacting objects. An example property with two objects is that a `Reader` should not be used after its `InputStream` has been closed. Supporting the verification of such properties is very important for checking the usage of real world APIs. However, the current design of our typestate analysis verifies each individual object based on its typestate and does not account for the a cluster of objects sharing the same state. This drawback limits the utility of this client. There have been some recent work to analyze typestate properties with multiple objects [4, 53]. We plan to review the existing techniques and eventually adapt the one that fits into the dynamic analysis setting.

7.3 Tracing Allocation Sites

Allocation sites are program locations that allocate objects at runtime. Tracking allocation sites is useful for improving software reliability and performance. For example, memory error detectors [19] can report the the allocation sites associated with the errors to facilitate debugging. Garbage collections can also benefit from allocation site analysis to perform the *pre-tenuring* optimization [14, 39]. Most past studies focus on reducing the space and time overhead of tracking allocation

sites using sampling- and probabilistic-based approaches [3, 19, 54, 40]. Whereas, MTF can also reduce the overhead of allocation site analysis because it supports fine-grained specification of the code regions that are most relevant to the analysis instead of tracking the whole program. In addition, context-sensitivity can be achieved by incorporating the PCC analysis discussed in Chapter 5. As a future work, we will develop such *selective allocation site analysis with context-sensitivity* client and integrate it into existing profiling APIs, e.g., JVMTI and DTrace to further improve program observability.

Chapter 8

Conclusions

Observability is useful for improving program understanding, reliability, and performance. In this thesis, we present a novel framework, Marker Tracing Framework (MTF), to improving program observability for virtual machine based languages. MTF provides a solid infrastructure for developing fine-grained trace-based dynamic program analyses. MTF allows the users to precisely specify code regions with special markers that can raise runtime events, which are received and propagated to each analysis client. The semantics of events are independently defined by each analysis such that multiple analysis clients can handle the events simultaneously with different logics. An extensible utility is also developed for adding marker instrumentation into the programs. In addition, new instrumentation client can be easily added into the utility to support new analyses.

Based on MTF, we have developed two analysis clients, i.e., typestate analysis and selective probabilistic calling context analysis (SPCC). The typestate analysis uses finite-state automaton to represent the typestate property and uses MTF to trace state transitions at runtime. By performing analysis inside the VM, it can extend the object header as well as collaborating with garbage collector for efficient and accurate

typestate verification. SPCC generalizes the existing PCC analysis with the ability to selectively computing PCC values for a set of methods, which can also be updated adaptively by the SPCC client based on the analysis results. Thus, we claim that MTF can offer great flexibility and adaptivity that collectively improve the precision and efficiency of such existing program analyses.

We have implemented both the framework and the analysis clients on the industry strength HotSpot JVM on the `x86` platform. Experiment results indicate that our MTF-enabled JVM offers sufficient runtime supports for both analysis clients with acceptable overhead. In the future work, we plan to further reduce the overhead by adopting existing techniques, e.g., static analysis and sampling. Moreover, several improvements are being evaluated to enhance the utility of the framework and the clients. Lastly, we propose to develop a selective allocation site analysis with context-sensitivity to further improve program observability.

Bibliography

[1] DTrace Probes in HotSpot VM, Last Retrieved: June 2010. Retrieved from http://java.sun.com/javase/6/docs/technotes/guides/vm/dtrace.html.

[2] JVM Tool Interface, Version 1.0, November 2004. Retrieved from http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html.

[3] Ole Agesen and Alex Garthwaite. Efficient object sampling via weak references. In *ISMM '00: Proceedings of the 2nd international symposium on Memory management*, pages 121–126, New York, NY, USA, 2000. ACM.

[4] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. *SIGPLAN Not.*, 40(10):345–364, 2005.

[5] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing jalapeno in java. *SIGPLAN Not.*, 34(10):314–324, 1999.

[6] Danilo Ansaloni, Walter Binder, Alex Villazón, and Philippe Moret. Parallel dynamic analysis on multicores with aspect-oriented programming. In *AOSD*

'10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development, pages 1–12, New York, NY, USA, 2010. ACM.

[7] Danilo Ansaloni, Walter Binder, Alex Villazón, and Philippe Moret. Rapid development of extensible profilers for the java virtual machine with aspect-oriented programming. In *WOSP/SIPEW '10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 57–62, New York, NY, USA, 2010. ACM.

[8] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. *SIGPLAN Not.*, 36(5):168–179, 2001.

[9] Matthew Arnold, Martin Vechev, and Eran Yahav. QVM: an efficient runtime for detecting defects in deployed systems. *SIGPLAN Not.*, 43(10):143–162, 2008.

[10] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: online modelling and performance-aware systems. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 15–15, Berkeley, CA, USA, 2003. USENIX Association.

[11] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 301–320, New York, NY, USA, 2007. ACM.

[12] Walter Binder, Jarle Hulaas, and Philippe Moret. Advanced java bytecode instrumentation. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 135–144, New York, NY, USA, 2007. ACM.

[13] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. ACM.

[14] Stephen M. Blackburn, Matthew Hertz, Kathryn S. Mckinley, J. Eliot B. Moss, and Ting Yang. Profile-based pretenuring. *ACM Trans. Program. Lang. Syst.*, 29(1):2, 2007.

[15] Eric Bodden. Efficient and Expressive Runtime Verification for Java. In *Grand Finals of the ACM Student Research Competition 2005*, 2005. Winner paper of the Grand Finals.

[16] Eric Bodden. Efficient Hybrid Typestate Analysis by Determining Continuation-Equivalent States. In *International Conference of Software Engineering (ICSE)*. ACM Press, 2010. To appear. Acceptance rate: 52/380 (13.7

[17] Eric Bodden, Laurie Hendren, Patrick Lam, Ondrej Lhotak, and Nomair A. Naeem. Collaborative Runtime Verification with Tracematches. *Oxford Journal of Logics and Computation*, 2008.

[18] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *SIGSOFT '08/FSE-16:*

Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, pages 36–47, New York, NY, USA, 2008. ACM.

[19] Michael D. Bond and Kathryn S. McKinley. Bell: bit-encoding online memory leak detection. *SIGOPS Oper. Syst. Rev.*, 40(5):61–72, 2006.

[20] Michael D. Bond and Kathryn S. McKinley. Probabilistic calling context. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 97–112, New York, NY, USA, 2007. ACM.

[21] S. Boroday, A. Petrenko, J. Singh, and H. Hallal. Dynamic analysis of java applications for multithreaded antipatterns. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7, New York, NY, USA, 2005. ACM.

[22] Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 569–588, New York, NY, USA, 2007. ACM.

[23] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 364–376, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[24] Markus Dahm. Byte Code Engineering with the BCEL API. Technical report, Freie University Berlin, April 2001.

[25] David Lutterkort. Finite Automata, 2010. Retrieved from http://augeas.net/libfa/.

[26] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 78–95, New York, NY, USA, 2003. ACM.

[27] Matthew B. Dwyer, Alex Kinneer, and Sebastian Elbaum. Adaptive Online Program Analysis. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 220–229, Washington, DC, USA, 2007. IEEE Computer Society.

[28] Matthew B. Dwyer and Rahul Purandare. Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 124–133, New York, NY, USA, 2007. ACM.

[29] Matthew B. Dwyer and Rahul Purandare. Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 124–133, New York, NY, USA, 2007. ACM.

[30] Romain Lenglet Eric Bruneton and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Proceedings of the ASF (ACM SIGOPS France) Journees Composants 2002: Adaptable and extensible component systems*, November 2002.

[31] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 133–144, New York, NY, USA, 2006. ACM.

[32] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[33] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[34] Robert Griesemer. Generation of virtual machine code at startup. In *OOPSLA '99 Workshop on Simplicity, Performance, and Portability in Virtual Machine Design*, 1999.

[35] Clemens Hammacher, Kevin Streit, Sebastian Hack, and Andreas Zeller. Profiling java programs for parallelism. In *IWMSE '09: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, pages 49–55, Washington, DC, USA, 2009. IEEE Computer Society.

[36] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer. Automating vertical profiling. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 281–296, New York, NY, USA, 2005. ACM.

[37] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. *SIGPLAN Not.*, 40(6):143–153, 2005.

[38] Gabriela Jacques-Silva, Roberto Jung Drebes, Júlio Gerchman, and Taisy Silva Weber. FIONA: A Fault Injector for Dependability Evaluation of Java-Based Network Applications. In *NCA '04: Proceedings of the Network Computing and Applications, Third IEEE International Symposium*, pages 303–308, Washington, DC, USA, 2004. IEEE Computer Society.

[39] Richard E. Jones and Chris Ryder. A study of java object demographics. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 121–130, New York, NY, USA, 2008. ACM.

[40] Maria Jump, Stephen M. Blackburn, and Kathryn S. McKinley. Dynamic object sampling for pretenuring. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 152–162, New York, NY, USA, 2004. ACM.

[41] Brian W. Kernighan and Rob Pike. *The practice of programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[42] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[43] Goh Kondoh and Tamiya Onodera. Finding bugs in java native interface programs. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 109–118, New York, NY, USA, 2008. ACM.

[44] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot$^{\text{TM}}$client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1):1–32, 2008.

[45] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking for data structure consistency. In *In 6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.

[46] Byeongcheol Lee, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. Debug all your code: portable mixed-environment debugging. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 207–226, New York, NY, USA, 2009. ACM.

[47] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[48] David Lo and Shahar Maoz. Hierarchical inter-object traces for specification mining. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 761–762, New York, NY, USA, 2008. ACM.

[49] Allen Davis Malony. *Performance observability*. PhD thesis, Champaign, IL, USA, 1990. Adviser-Reed, D.

[50] Marcus Meyerhöfer. TestEJB: response time measurement and call dependency tracing for ejbs. In *MAI '07: Proceedings of the 1st workshop on Middleware-application interaction*, pages 55–60, New York, NY, USA, 2007. ACM.

[51] Microsoft Corporation. Common Object File Format (COFF), 2006. Retrieved from http://support.microsoft.com/?id=121460.

[52] Todd Mytkowicz, Devin Coughlin, and Amer Diwan. Inferred call path profiling. *SIGPLAN Not.*, 44(10):175–190, 2009.

[53] Nomair A. Naeem and Ondrej Lhotak. Typestate-like analysis of multiple interacting objects. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 347–366, New York, NY, USA, 2008. ACM.

[54] Rei Odaira, Kazunori Ogata, Kiyokuni Kawachiya, Tamiya Onodera, and Toshio Nakatani. Efficient runtime tracking of allocation sites in java. In *VEE '10: Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 109–120, New York, NY, USA, 2010. ACM.

[55] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspottm server compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[56] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspottm server compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[57] Alexandar Pantaleev and Atanas Rountev. Identifying Data Transfer Objects in EJB Applications. In *WODA '07: Proceedings of the 5th International Workshop on Dynamic Analysis*, page 5, Washington, DC, USA, 2007. IEEE Computer Society.

[58] David J. Pearce, Matthew Webster, Robert Berry, and Paul H. J. Kelly. Profiling with aspectj. *Softw. Pract. Exper.*, 37(7):747–777, 2007.

[59] Matt Pietrek. The .NET Profiling API and the DNProfiler Tool. *MSDN Magazine*, December 2001. Retrieved from http://msdn.microsoft.com/en-us/magazine/cc301725.aspx.

[60] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 535–552, New York, NY, USA, 2007. ACM.

[61] Steven P. Reiss. Controlled dynamic performance analysis. In *WOSP '08: Proceedings of the 7th international workshop on Software and performance*, pages 43–54, New York, NY, USA, 2008. ACM.

[62] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: adaptive selection of collections. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 408–418, New York, NY, USA, 2009. ACM.

[63] Antonio Da Silva, Alberto Gonzalez-Calero, José Fernán Martinez, Lourdes Lopez, Ana Belen Garcia, and Vicente Hernandez. Design and Implementation of a Java Fault Injector for Exhaustif®SWIFI Tool. In *DEPCOS-RELCOMEX '09: Proceedings of the 2009 Fourth International Conference on Dependability of Computer Systems*, pages 77–83, Washington, DC, USA, 2009. IEEE Computer Society.

[64] Paramvir Singh and Hardeep Singh. Dynametrics: a runtime metric-based analysis tool for object-oriented software systems. *SIGSOFT Softw. Eng. Notes*, 33(6):1–6, 2008.

[65] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[66] William N. Sumner, Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang. Precise calling context encoding. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 525–534, New York, NY, USA, 2010. ACM.

[67] Sun Microsystems, Inc. Java™Debug Interface, 2004. Retrieved from http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi/.

[68] Nikolai Tillmann and Jonathan de Halleux. Pex - White Box Test Generation for .NET. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, chapter 10, pages 134–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[69] TIS Committee. Executable and Linking Format Specification Version 1.2, May 1995. Retrieved from http://refspecs.freestandards.org/elf/elf.pdf.

[70] David Ungar, Adam Spitz, and Alex Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 11–20, New York, NY, USA, 2005. ACM.

[71] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[72] Alex Villazón, Walter Binder, Danilo Ansaloni, and Philippe Moret. Advanced runtime adaptation for java. In *GPCE '09: Proceedings of the eighth international conference on Generative programming and component engineering*, pages 85–94, New York, NY, USA, 2009. ACM.

[73] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing java programs. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 512–521, Washington, DC, USA, 2004. IEEE Computer Society.

[74] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *IWMM '92: Proceedings of the International Workshop on Memory Management*, pages 1–42, London, UK, 1992. Springer-Verlag.

[75] Feng Xian, Witawas Srisa-an, and Hong Jiang. Allocation-phase aware thread scheduling policies to improve garbage collection performance. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 79–90, New York, NY, USA, 2007. ACM.

[76] Feng Xian, Witawas Srisa-an, and Hong Jiang. Contention-aware scheduler: unlocking execution parallelism in multithreaded java programs. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 163–180, New York, NY, USA, 2008. ACM.