Performance Measurement of Interpreted, Just-in-time Compiled, and Dynamically Compiled Executions

BY

Tia Newhall

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the University of Wisconsin—Madison

1999

Acknowledgments

During the course of my graduate career I have benefited from the help, support, advice and suggestions of a great many people.

My thesis advisor, Bart Miller, provided years of technical and professional guidance, advice, and support. I am grateful for all that he has taught me, both as a researcher and as a teacher.

I thank my committee members Marvin Solomon and Miron Livny for their time and effort spent reading my thesis, and for providing valuable criticisms and suggestions for improving my work. I thank my entire committee, Marvin Solomon, Miron Livny, Mary Vernon, and Douglas Bates, for a lively and thorough discussion of my thesis work during my defense.

I am grateful I had the opportunity to be a member of the Paradyn project, and in particular for the stimulating interactions with Paradyn project members. I am indebted to the help and support I have received from Zhichen Xu, Ari Tamches, Vic Zandy, Bruce Irvin, Mark Callaghan, Marcelo Goncalves, Brian Wylie and all the other current and past members of the Paradyn project. A special thanks to Karen Karavanic who has been a supportive comrade at each step in this process.

I am indebted to NCR for the support provided by a graduate research fellowship during my final year as a graduate student. Also, I would like to acknowledge the groups that have helped fund my research assistantships: Department of Energy Grant DE-FG02-93ER25176, NSF grants CDA-9623632 and EIA-9870684, and DARPA contract N66001-97-C-8532.

I thank Marvin Solomon and Andrew Prock for providing the Java application programs used for the performance measurement studies in this dissertation.

My graduate school friends have been a constant source of moral support, advice, and amusement. In particular, I'd like to thank Mary Tork-Roth, Bill Roth, Kurt Brown, Brad Richards, Holly Boaz, Mark Craven, Susan Goral, Susan Hert and Alain Kägi.

Most of all, I would not have been able to accomplish my goals without the love, support and encouragement of Martha Townsend and the rest of my family. I dedicate this work to them.

Contents

List of Figures

Chapter 1

Introduction

With the increasing popularity of Java, interpreted, just-in-time compiled and dynamically compiled executions are becoming a common way for application programs to be executed. As a result, performance profiling tools for these types of executions are in greater demand by program developers. In this thesis, we present techniques for measuring and representing performance data from interpreted, just-in-time and dynamically compiled program executions. Our techniques solve problems related to the unique characteristics of these executions that make performance measurement difficult, namely that there is an interdependence between the interpreter program and the application program, and that application program code is transformed at runtime by just-in-time and dynamic compilers.

1.1 Motivation

An *interpreted execution* is the execution of one program (the application) by another (the interpreter) in which the interpreter implements a virtual machine that takes the application as input and runs it. Interpreters act as runtime translators; program code that is targeted to run on the virtual machine is translated to run on the host machine. Examples include interpreters for programs written in LISP[46], Prolog[38], Basic[36], Scheme[34], Smalltalk[18], Perl[69], TCL[48], Pascal[59], Python[68], Self[29], and Java[41]. One benefit of interpreted execution is the platform independence of the application program; an application program can run on any machine on which the interpreter virtual machine runs. Another benefit of interpreted execution is that it can be used to emulate systems or parts of systems that may not be present on the underlying operating system/architecture of the host machine; for example, simulator programs such as SimOS [57], g88 [5], FAST [6], RSIM [49], WWT [56], and Shade [13] implement a virtual machine, and they take as input and run application programs that are targeted to run on the simulated architecture.

Typically, interpreted executions are orders of magnitude slower than equivalent native exe-

cutions [44, 58]. A faster way to run the application is to translate large parts (like entire functions) to native code, and directly execute the cached translation rather than interpreting the application one instruction at a time. Just-in-time (JIT) compilers [16, 42] and dynamic compilers [3, 14, 29] execute applications in this manner. Also, many fast interpreters [13, 57, 18, 52, 42] do some translating and caching of application code.

Interpreted, JIT compiled, and dynamically compiled program executions are increasingly being used as the norm in running applications. For example, Java applications are almost always run by a Java virtual machine that is implemented as an interpreter [41], JIT compiler [16] or dynamic compiler [23, 63, 9]. The platform independence of Java programs, and the ability to attach Java programs to web pages combined with the increasing popularity of the world wide web, have contributed to the use of Java for various types of applications including parallel and distributed computing [19, 67, 12], and Web-based computing and meta-computing [4, 8, 20]; Java is increasingly being used for large, complex applications. As a result, Java programmers are becoming more concerned with their program's performance, and thus have more of a need for performance measurement tools that can help them answer questions about their program's performance. Therefore, being able to build performance measurement tools for interpreted, JIT compiled, and dynamically compiled executions will become increasingly important. However, there are two unique characteristics of these types of executions that make performance measurement difficult. First, in interpreted executions there is an interdependence between the interpreter's execution and the interpreted application's execution. Second, performance measurement of dynamically compiled and JIT compiled executions is complicated by the application program's multiple execution forms.

In this thesis, we discuss techniques for collecting and representing performance data from interpreted, JIT compiled and dynamically compiled program executions that solve problems associated with the unique characteristics of these types of executions that make performance measurement difficult: (1) the interdependence between the interpreter's and the interpreted application's execution, and (2) the multiple execution forms of JIT compiled and dynamically compiled application code that is translated at run-time by the virtual machine.

1.1.1 Performance Measurement of Interpreted Executions

The interdependence between the execution of the interpreter and the interpreted code makes performance measurement difficult. The implementation of the interpreter determines how application code is executed and constructs in the application trigger the execution of specific code in the interpreter. The difficulties this causes are illustrated by comparing an interpreted code's execution to a compiled code's execution (shown in Figure 1.1). A compiled code is in a form that can be executed directly on a particular operating system/architecture platform. Tools that measure the performance of the execution of a compiled code provide performance measurements in terms of platform-specific costs associated with executing the code; process time, number of page faults,

I/O blocking time, and cache miss rate are some examples of platform-specific measures. In contrast, an interpreted code is in a form that can be executed by the interpreter. The interpreter virtual machine is itself an application program that executes on the OS/architecture platform. One obvious difference between compiled and interpreted application execution is the extra layer of the interpreter program that, in part, determines the performance of the interpreted application. We call the Interpreter layer the *virtual machine* (VM) and the Application layer the *application program* (AP). The VM is any program that implements a virtual machine for another application that it takes as input and runs.

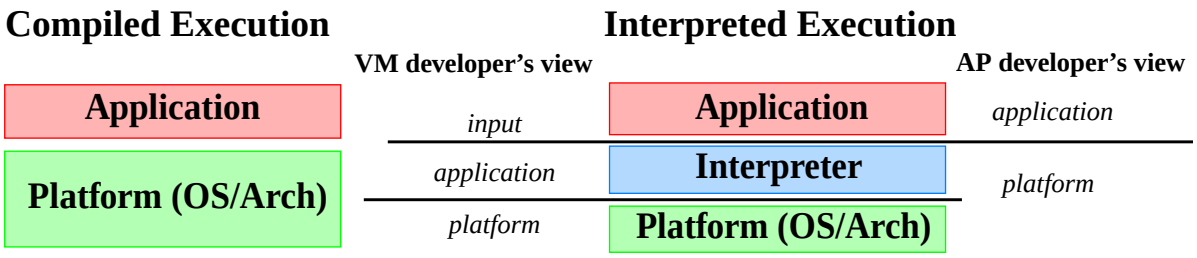

Figure 1.1 Compiled application's execution vs. Interpreted application's execution. *A VM developer and an AP developer view the interpreted execution differently.*

Performance data that explicitly describes the interaction between the virtual machine and the application program is critical to understanding the performance of the interpreted execution, and will help a program developer more easily determine how to tune the application to make it run faster. For example, if performance data shows that the amount of VM overhead associated with the VM interpreting `call` instructions in AP method `foo` accounts for a large fraction of `foo`'s execution time, then the program developer knows that one way to improve `foo`'s interpreted execution is to reduce some of this VM overhead in `foo`'s execution.

Because there is an Application layer and an Interpreter layer in an interpreted execution, there are potentially two different program developers who would be interested in performance measurement of the interpreted execution: the VM developer and the AP developer. Both want performance data described in terms of platform-specific costs associated with executing parts of their applications. However, each views the platform and the application program as different layers of the interpreted execution. The VM developer sees the AP as input to the VM (as shown in the second column of Figure 1.1). The AP developer sees the AP as a program that is run on the virtual machine implemented by the VM (shown in the last column of Figure 1.1).

The VM developer is interested in platform-specific performance measurements associated with the virtual machine's execution and characterized in terms of its input (the AP); the VM developer wants performance data that characterizes the VM's performance in terms of the application code it interprets. An example of this type of performance data is the amount of process time used while VM function `objectCreate` is interpreting instructions from an AP method.

The AP developer, on the other hand, views the platform as the virtual machine implemented by the interpreter program. An AP developer wants VM-specific performance measurements that allow an AP developer to see inside the virtual machine to understand the fundamental costs associated with the virtual machine's execution of the application. An example of this type of performance data is the amount of VM object creation overhead in the execution of AP method `foo`. A performance tool must present performance data that describes specific VM-AP interactions in a language that matches the program developer's view of the execution.

Our approach can address any environment where one program runs another. The machine hardware can be viewed as running the operation system that runs the user program. The part of our solution for describing performance data for interacting VM and AP layers is applicable to describing performance data associated with interactions between multiple levels.

1.1.2 Performance Measurement of Application's with Multiple Execution Forms

Performance measurement of JIT and dynamically compiled application programs is difficult because of the application's multiple execution forms (AP code is transformed into other forms while it is executed). For example, a Java program starts out interpreted in byte-code form. While it is executed, a Java dynamic compiler VM may translate parts of the byte-code to native code that is directly executed on the host machine. Figure 1.2 shows the two execution modes of an environment that uses dynamic compilation to execute an AP that starts out in byte-code form: (1) the VM interprets AP byte-codes; (2) native code versions of AP methods that the VM compiles at runtime are directly executed by the operating system/architecture platform with some residual VM interaction (for example, activities like object creation, thread synchronization, exception handling, garbage collection, and calls from native code to byte-code methods may require VM interaction). The VM acts like a runtime library to the native form of an AP method. At any point in the execution, the VM may compile a method, while some methods may never be compiled.

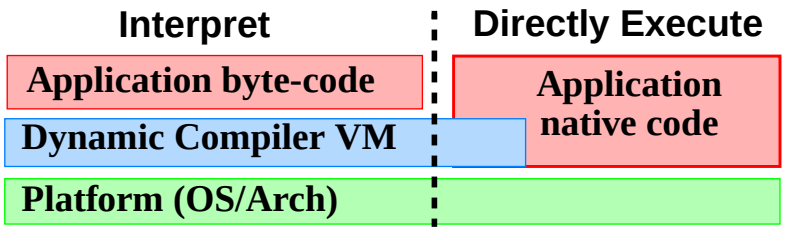

Figure 1.2 Dynamic Compilation of AP byte-codes. During a dynamically compiled execution methods may be interpreted by the VM and/or compiled into native code and directly executed. *The native code may still interact with the VM; the VM acts like a runtime library to the AP.*

Because parts of the application program are transformed from one form to another by the

VM at runtime, the location and structure of application code can change during execution. From the standpoint of performance measurement, this causes several problems. First, a performance tool must measure each form of the Java method, requiring different types of instrumentation technologies. Second, a tool must be aware of the relationship between the byte-code and native code version of a method, so that performance data can be correlated. Third, because AP code is transformed at run-time, its transformation is part of the execution; a performance tool must represent performance data associated with the transformational activities. Finally, since even the native code methods interact with the VM (with the VM acting more like a run-time library), performance data that explicitly describes these VM interactions with the native form of an AP method will help a programer better understand the application's execution.

1.2 Summary of Results

This dissertation presents new methods for collecting and representing performance data for interpreted, JIT compiled, and dynamically compiled executions. We present a representational model for describing performance data from these types of executions that addresses problems associated with the interdependence between the execution of the AP and the VM, and addresses problems associated with the multiple execution forms of the AP. We show how a performance tool based on our model provides performance data that allows program developers to easily determine how to tune their programs to improve the program's performance. We demonstrate the effectiveness of the model by using performance data from our tool to improve the execution times of several Java application programs.

Our model allows for a concrete description of behaviors in interpreted, JIT compiled and dynamically compiled executions, and it is a reference point for what is needed to implement a performance tool for measuring these types of executions. An implementation of our model can answer performance questions about specific interactions between the VM and the AP, and it can represent performance data in a language that both an application program developer and a virtual machine developer can understand. The model describes performance data in terms of the different forms of an application program object, describes run-time transformational costs associated with dynamically compiled AP code, and correlates performance data collected for one form of an AP object with other forms of the same object.

We present Paradyn-J, a prototype performance tool for measuring interpreted and dynamically compiled Java executions. Paradyn-J is an implementation of our model for describing performance data from these types of executions. To demonstrate our ideas, we show how Paradyn-J describes performance data that can be only represented by tools based on our model, and how this data provides information that is useful to both an AP developer and a VM developer in determining how to tune the program to make it run faster. We present results using performance data from Paradyn-J to tune an all-interpreted Java CPU simulator program. Performance data from Paradyn-J identifies expensive Java VM activities (method call and object creation over-

head), and represents specific VM costs associated with constrained parts of the Java application. With this data we were easily able to determine how to tune the Java application to improve its performance by a factor of 1.7. We also present results using Paradyn-J to tune a dynamically compiled Java neural network application. In this performance tuning study, we simulate the dynamic compilation of several application methods. Paradyn-J provides performance measures associated with the byte-code and native code forms of the dynamically compiled methods, and in particular, measures VM object creation overhead associated with each form of the method. These data allow us easily to determine how to tune one of the dynamically compiled methods to improve its performance by 10%.

1.3 Organization of Dissertation

This dissertation is organized into seven chapters. We begin by discussing related work in Chapter 2.

In Chapter 3 we present the first part of our solution: our model for describing performance data from interpreted executions. Chapter 4 presents Paradyn-J, a performance tool for measuring interpreted Java executions that is based on our model. We describe Paradyn-J's implementation, and demonstrate our ideas by showing how performance data from Paradyn-J can be used to improve the performance of a Java CPU simulator application.

We present results from a performance study comparing dynamically compiled and interpreted Java application executions in Chapter 5. We use this study to motivate the need for detailed performance data from dynamically compiled executions.

In Chapter 6, we describe the second part of our solution: a model for describing performance data from program executions that have multiple execution forms. We describe modifications to Paradyn-J for measuring a simulation of dynamically compiled Java executions, and demonstrate our model by using performance data from Paradyn-J to tune a dynamically compiled method function from a Java neural network application.

Chapter 7 discusses implementation issues particular to implementing a performance tool based on our model. We also discuss some of the complexities of our implementation of Paradyn-J and examine some other ways in which a performance tool implementing our model could be built.

Finally, in Chapter 8, we present our conclusions, and suggest future directions for this work.

Chapter 2

Related Work

Past research in application-level performance measurement tools has addressed many issues related to the measurement of interpreted, JIT compiled and dynamically compiled program executions. In particular, there are several performance measurement tools for JIT compiled and interpreted executions that provide useful information to an AP developer, including a few that provide a limited number of fixed VM costs (such as number of object creates, and counts of garbage collections activities). However, there is no tool that can describe performance data for general interactions between the VM and the AP (such as VM method call overhead due to calls made from a particular AP method); no existing tool exports VM resources and, as a result, arbitrary, general performance measurement of the VM and of VM-AP interactions is not possible using existing tools. Also, there is no tool that we know of that can correlate performance data for the different execution forms of dynamically compiled AP code.

We show in later chapters that having performance data that measure specific VM costs associated with AP's execution, and that measure AP's multiple execution forms, is critical to understanding the performance of interpreted, JIT or dynamically compiled executions. To provide this type of performance data there are several issues that we must address. First, we must obtain performance measures from both the VM and the AP's execution. Second, we must extract abstractions implemented by the VM program (e.g., AP threads) and measure performance data in a way that is consistent with these abstractions. Finally, we must obtain mapping information when AP code is compiled at run-time and use this information to map performance data between the different execution forms of AP code. Previous work in performance measurement tools has addressed similar issues.

We must explicitly measure the VM program. Any general purpose performance tool can provide measures of the VM program. However, we must also be able to describe VM measures in terms of the AP-developer's view of the execution to allow the AP developer to see inside the VM. To do this we need to expose abstractions implemented by the VM in a language that the AP

developer can understand (in terms of the abstractions that the VM implements). Similar work has been done to allow a binary application developer to see inside the operating system, giving a view of how the OS executes an application, and to see inside user level libraries to present performance data in terms of the abstractions implemented by the library.

We need to map performance data between different views of AP code that changes form at run-time. All performance tools do some type of mapping between low-level execution activities and high-level views of these activities; the most common is mapping performance data to a source code view. There are also research efforts examining more complicated mappings, such as performance measurement for data parallel programs that map low-level synchronization activities to language-level views of these activities.

In this chapter, we discuss systems that address pieces of the problems we need to solve. We conclude that although there are currently tools that can provide some of the features we want, there are no tools, or combinations of tools, that can explicitly represent arbitrary, general VM-AP interactions from interpreted, JIT compiled, or dynamically compiled executions. Also, there are no tools that can describe performance data in terms of both the AP-developer's and the VM-developer's view of the execution, and there are no tools that can represent performance in terms of different execution forms of dynamically compiled AP code, nor represent costs associated with the run-time compilation of AP code.

2.1 Performance tools for interpreted and JIT compiled executions

There are several performance profiling tools for measuring interpreted and JIT compiled applications. These tools provide performance data in terms of the application's execution. Some tools instrument AP source code prior to the AP's execution. When run by the VM, AP instrumentation code is interpreted just like any other AP code. Tools implemented in this way provide no performance measurement of the VM's execution. Other tools are implemented as special versions of the VM or interact with the VM at run-time using VM API's to obtain performance measures of the AP's execution; these tools have the potential to obtain measures of both the VM and AP's execution.

Tools that instrument the application source code or byte-code prior to execution by the VM include the Metering Lisp profiling tool [35], NetProf [50] and ProfBuilder [15]. NetProf and ProfBuilder re-write the Java .class files by inserting calls to instrumentation library routines. When the modified application code is run by the VM, an instrumentation library collects timing information associated with the execution of the instrumented application code. Because these tools instrument Java .class files, they can easily obtain fine-grained performance measures, such as basic-block or statement level performance measures. Also, the Java byte-code instrumenting tools do not need application source code to measure the Java application.

Inserting instrumentation in the application prior to its execution, and letting the VM execute the instrumentation code along with the other instructions in the application, is an easy way to obtain timing and counting measures in terms of the application's code, but there are several problems with this approach. First, there is no way to know which VM activities are included in timing measures; timing measures associated with a method function could include thread context switching[1], Java class file loading, garbage collection and run-time compilation. Second, there is no way of obtaining measurements that describe specific VM overheads associated with VM's execution of the application, since these measures require instrumenting VM code. Finally, for JIT compiled and dynamically compiled executions, there is no control over how the compiler transforms the instrumentation code; the compiler could perform optimizations that re-order instrumentation code and method code instructions in such a way that the instrumentation code is no longer measuring the same thing it was prior to compilation.

There are tools for measuring interpreted and JIT compiled Java programs that provide some measure of Java VM costs associated with the application's execution. To obtain these measures, the tools are either implemented as special versions of the Java VM (JProbe [37], JDK's VM [60], Visual Quantify [53], and Jinsight [30]), or they interact with the Java VM at run-time using API's implemented by the VM (OptimizeIt [32], and VTune[31]). Also, there is a new Java profiling interface (JVMPI [62]) with Sun's Java 2 Platform [63]. JVMPI can be used to build performance measurement tools that can obtain certain information about how the Java VM runs the Java application.

Special versions of the Java VM contain profiling code that is executed as part of the VM's interpreter or JIT compiler code. The profiling code obtains counting and timing measures in terms of the Java AP code that is being executed. One example is Sun's JDK VM implementation with built-in profiling. Instrumentation code, which is part of the VM code, can be triggered to collect the total execution time and the number of calls to Java application methods, as well as to create a call-graph. Another example is JProbe for interpreted and JIT compiled Java executions. It provides a call graph display with cumulative CPU time and count measures for methods, a real time memory usage display showing allocated and garbage collected object space and object instance counts, and a source code display that is annotated with total CPU and count measures. Rational's Visual Quantify is a performance tool for interpreted Java executions that is integrated with Microsoft's Developer Studio 97. It provides a call graph display and a source code display where lines of source code are annotated with total execution times and number of executions. It also provides a real-time display of active Java application threads. Finally, Jinsight is a traced-based performance tool that gathers trace data from a modified Java VM running on Microsoft's Windows or IBM's AIX operating systems. It provides displays for viewing performance data in terms of object creates, garbage collection, and execution sequences of Java code, and provides a display of real-time application thread interaction.

1. The timing instrumentation used by these tools is not thread aware.

There are two Java profiling tools that interact with the Java VM at run-time using an API implemented by the VM to obtain performance data. One example is Intel Corporation's VTune. VTune uses event-based sampling, and hardware performance counters available on Intel's Pentium Pro processors to collect performance data for JIT compiled Java applications. VTune runs on Intel's Pentium Processor based platforms running versions of Microsoft's Windows operating systems. The tool handles processor-event interrupts to obtain hardware performance counter and program counter samples. It interacts with the Java VM through a special VTune API; only if the VM implements the VTune API can VTune associate performance data with Java AP code. Currently, Microsoft's Visual J++ and Internet Explorer, Borland's JBuilder 2, and Asymetrix's SuperCede support the VTune API. The API is used to determine where JIT compiled Java application code is located in memory, so that the program counter samples can be correlated with Java application code. VTune provides displays of total CPU time, and number of calls associated with Java methods. It also provides a call graph display.

Another example of a tool that interacts with the Java VM at run-time is Intuitive Systems' OptimizeIt. OptimizeIt is a tool for measuring interpreted and JIT compiled Java executions run by Sun's unmodified Java VM for versions of JDK up to the Java 2 Platform release. OptimizeIt provides total CPU time for each Java application thread, total CPU time associated with application methods, and a real time memory profiler that provides the number of object instances per class. It also provides a display that correlates total CPU times for application threads with individual lines of Java application source code. OptimizeIt starts all Java applications that it measures. At start-up, it interacts with the Java VM to initiate performance monitoring. For the 1.1 versions of JDK, it uses low-level API's in the Java VM to obtain performance data for the application. Also, it may need to force the run-time linker to load shared object files containing special versions of Java VM routines that are used to obtain some VM information. Because OptimizeIt interacts with the Java VM at run-time, it has to be ported to different versions of the VM. However, the March 1999 release of OptimizeIt is moving towards a more platform-independent implementation by using new classes and the new JVMPI interface available with Sun's Java 2 Platform to obtain its profiling data.

JVMPI is an API available with Sun's Java 2 Platform that implements an interface for performance measurement tools. JVMPI is a two-way function call interface between the Java VM and a profiler agent in the VM process. A performance tool designer builds a profiling tool by implementing the profiler agent code that interacts with the VM using calls and event callbacks to obtain performance information. The profiler agent is written in C or C++ code using the Java Native Interface (JNI [61]) to call JVMPI functions. A performance tool designer also implements a front-end part of the tool that obtains performance measures from the profiler agent and displays them to the user. Through JVMPI, the Java VM exports information about some of its abstractions such as per thread CPU time, synchronization activities, object creates, and class file loading. It also exports information about the Java application code it runs. Using this interface, a

platform-independent tool can be built that provides some VM costs and run-time activities associated with the application's execution; the performance tool will run on any VM that implements the JVMPI interface.

All of these profiling tools represent performance data in term of the interpreted or JIT compiled application's execution. Some of these tools provide measures of specific VM costs associated with the application's execution. For example, JProbe, OptimizeIt and Visual Quantify provide some type of memory profiling information associated with Java method functions. This type of performance data helps an application developer to more easily answer questions of how to tune the application to improve its interpreted or JIT compiled performance.

2.2 Traditional performance tools

There are many general purpose performance tools for parallel and sequential programs that provide performance measures in terms of the program's execution. Most of these tools for software measurement probes to be inserted in an application program's source code [43], or inserted by re-compiling or re-linking with an instrumentation library [55, 73, 70, 22, 54], or by re-writing the binary [40], or dynamically at run-time [47]. Most of these tools can be classified as either profiling [22, 70, 43, 54] or event-tracing tools [55, 73, 43, 25].

Profiling tools typically insert instrumentation code to count and/or time the execution of fragments of application code, run the application, and compute the total value of the performance metrics associated with different code fragments. One problem with profiling is that detailed information, such as time-varying data, can be lost in the summary data. Trace-based tools insert code in the application to generate a time-stamped record of events during the program's execution. As the program runs, trace events are written to a log. The performance tool analyzes trace data from the log and displays it to the user. Trace-based tool can capture very detailed information about an application's execution; however, for long-running or massively parallel programs, generating, storing, and analyzing the trace files becomes problematic.

Paradyn is a tool for measuring the performance of long-running, large, parallel applications. Paradyn is designed to solve some of the problems associated with profile and trace-based tools. The performance data collection part of the tool is designed to scale to long-running, massively parallel applications by using two mechanisms: fixed length data structures to store time varying performance data and dynamic instrumentation [27] (instrumentation code that can be inserted or removed from a running program at any point in its execution).

Digital's continuous profiling infrastructure [51] uses a slightly different technique to provide low-overhead performance data. It periodically samples the Alpha performance counter hardware to obtain profiling information: a device driver services interrupts from the Alpha performance counters; on each interrupt, the driver records the process identifier and program counter for the interrupted program; samples are buffered in the kernel, and periodically extracted by a daemon

process that stores them in an on-disk database. Performance tools can be written to extract information from this database.

2.3 Tools that Map Performance Data to User's View of Program

All performance measurement tools provide some mapping between a low-level and high-level view of an action. For example, most performance tools can associate performance measures of machine code instructions with a source code view of the performance data (a machine code to source code mapping). There are also tools that can provide more complicated hierarchical mappings between data parallel code and low-level runtime activities. For example, some tools for parallel Fortran codes are integrated with the compiler (MPP Apprentice [70] and FortranD [1]). The compiler generates mapping information that the tool uses to correlate performance measures, like synchronization times, with source code fragments or data parallel arrays; the execution of application binary code can be mapped to the application developer's view of the program.

Another example is the NV performance tool model [33] for measuring CM Fortran [66] programs. NV is designed to map performance data collected for the execution of low-level code to its high-level view in terms of parallel data structures and statements in data parallel Fortran programs. NV is not integrated with the Fortran compiler. Instead, it uses static information from the application's a.out file and information obtained at run-time to map between the execution of a high-level language statement and the low level actions that implement the statement's execution. MemSpy [45] is a data-oriented profiling tool that provides performance measures such as cache invalidation, and local and remote miss rates associated with parallel data objects. It works by using the Tango simulation and tracing system [17] to instrument applications. MemSpy maps the causes of cache misses in Tango's simulated caches to parallel data objects.

Techniques designed for debugging optimized code [7, 26] solve a similar problem. Here, the problem of accurately relating activities in the binary's execution to a portion of source code is complicated by compiler optimizations; there is not necessarily a one-to-one mapping between source code and binary code fragments. Setting breakpoints or stepping through the execution of the binary and mapping back to the source code is complicated by compiler optimizations that inline, replicate, or reorder the code.

2.4 Tools that can See Inside the Kernel

KiTrace [39] and KernInst [65] are examples of performance tools that instrument kernel code. They allow a user to see inside the operating system by providing performance data in terms of kernel code. For example, KernInst is a tool for instrumenting a commodity "off the shelf" kernel at run-time. KernInst allows a user to instrument kernel code at an instruction level granularity. KernInst provides an infrastructure to build profiling tools and provides an interface for modifying the kernel routines at run-time. Kernel profiling tools have focused on making the OS visible by providing performance measures of kernel routines, but have not explicitly focused on

associating OS activities with the application code that it executes; the performance measures are not explicitly correlated with the user-level application code of the workloads run on the OS.

2.5 Tools that Expose Abstractions from User-Level Libraries

There are tools that can expose abstractions implemented in user-level libraries to an application that uses the library. For example, there are tools that can provide performance data for user level thread libraries (CMON [10], and Thread Aware Paradyn [72]), or for distributed shared memory (Paradyn-Blizzard [71]). These tools make the abstractions implemented by the library visible to the user; the tool measures and describes performance data in a way that is consistent with the library's abstractions. For example, the thread aware version of Paradyn exports a thread view to the tool user so the user can see performance data associated with a particular user-level thread. The tool interacts with the thread library by turning on and off timer instrumentation based on which thread is executing particular parts of the application code, and based on when thread context switching occurs; the tool can correctly measure performance data in terms of individual application threads in the presence of thread context switching.

2.6 Conclusions

There are several performance measurement tools for measuring interpreted, and JIT compiled executions in terms of the application program. However, one problem present in all existing performance tools is that they do not support explicit measurement of the VM; explicit measurement of the VM is necessary to help define why the application performs the way it does. All these tools can provide a measure of the total time spent executing a Java method function, but none can provide a measure of specific Java VM activities that are included in this time. For example, a method may contain certain byte-code instruction sequences that are expensive for the VM to interpret, or its execution time may include method table lookups, garbage collection activities, class file loading, or thread context switches performed by the Java VM. A tool that can describe performance data that measures these specific VM costs associated with the method will help an AP developer to more easily understand the method's performance. Sun's new JVMPI interface has the potential to be used for obtaining some of this information. However, it currently falls short of providing explicit measurement of VM activities that can be associated with the execution of AP code. In Chapter 7, we propose changes to JVMPI that would result in making some of these data accessible to a tool builder.

Also, we know of no existing tools for measuring dynamically compiled Java. Since dynamic compiled Java executions have the potential to be as fast as, or faster than, equivalent C++ executions [23], we expect that Java applications will increasingly be run on virtual machines implemented as dynamic compilers. As a result, being able to build performance tools that can deal with run-time compiled Java code will become increasingly important. A final issue is that without exporting the VM's code, these tools are of little use to a VM developer.

In our work we use and build on many of the techniques used by traditional performance measurement tools, and by tools that: (1) map performance data to a high-level view of the data; and (2) describe performance data in terms of abstractions implemented by user-level libraries. We use similar techniques to: (1) map performance data between multiple executions forms of dynamically compiled AP code; and (2) describe performance data in terms of abstractions implemented by the VM (for example, threads, synchronization, garbage collection, and VM execution state).

In the remainder of this thesis, we show how tools that measure the VM, that measure the AP, that measure interactions between the VM and AP, and that can measure AP code that is compiled at run-time, have the potential to describe any VM-AP interaction in the execution in a language that both an AP and a VM developer can understand. As a result, such tools can describe performance data that is critical to understanding the performance of interpreted, JIT compiled, and dynamically compiled executions.

Chapter 3

Describing Performance Data that Represent VM-AP Interactions

We present a representational model for describing performance data from interpreted, JIT compiled, and dynamically compiled executions. We focus first on the part of the model that describes the interaction between the virtual machine and the application program during execution; we also focus on the part of the model that describes performance data in terms of both the virtual machine developer's and the application developer's view of the execution. As a result, we are able to represent performance data that describe specific VM-AP interactions in a language that either developer can understand.

3.1 Representing an Interpreted Execution

To describe performance data from interpreted, JIT compiled, and dynamically compiled executions, we first need to represent these types of executions. We extend Paradyn's representation of program resources and resource hierarchies for representing a single program's execution [47] to represent the execution of the AP by the VM. We first discuss a representation of the AP's and the VM's executions separately, and then combine their representations to produce the model of a virtual machine running an application program.

3.1.1 Representing a Program Execution

A running program can be viewed as a set of physical and logical components called program *resources*. For example, a running program contains processes, accesses pages of memory, executes its code, runs on a set of host machines, and may read or write to files. Each of these program objects can be represented by a resource. A process, function, semaphore, memory block, message tag, and file descriptor are all program resources. A *program execution* is represented by the set of its program resources.

Some program resources are related in a hierarchical manner. For example, a module resource consists of several function resources, and a function resource consists of several statement resources. By grouping related resources hierarchically, we represent a program execution

as a set of its program resource hierarchies. A *resource hierarchy* is a collection of related program resources, and can be thought of as a view of a running program in terms of these related program resources. For example, the Process resource hierarchy views the running program as a set of processes. It consists of a root node representing all processes in the application, and some number of child resources–one for each process in the program execution. Other examples of resource hierarchies are a Code hierarchy for the code view of the program, a Machine resource hierarchy for the set of hosts on which the application runs, and a Synchronization hierarchy for the set of synchronization objects in the application. An application's execution might be represented as the following set of resource hierarchies: `[Process,Machine,Code,SyncObj]`. Other possible hierarchies include Memory, and I/O.

We use an object-oriented representation to describe program resources. The common characteristics of all resources are represented by a Resource base class. Different types of resources are represented by classes derived from the Resource base class. For example, module, function, process, message tag, and machine are distinct types of resources that are represented by classes derived from the base classes. The Resource base class implements information that is common to all resources, such as the name of the resource, resource's parent resource instance, list of child resource instances, and a constraint method function(s). The constraint method is a boolean function that is true when the resource is active. We discuss constraint methods in more detail in Section 3.2.1.

The derived classes contain information that is unique to a particular resource type. For example, the **Function** class might have information about a function's address and size. Other examples of information unique to specific resource classes are shown in Figure 3.1.

We represent a program execution, **P**, by a list of its resource hierarchy root node resource instances. For example, **P** = `[Code,Process,SyncObj,File,Memory,Machine]` is the list of resource hierarchy root nodes for the program execution **P**, and **P**`[Code]` is root node resource instance of the Code hierarchy. We use the shorthand notation "`root_name`" for naming the resource instance "**P**`[root_name]`". For example, we will write `Code` for **P**`[Code]`. Figure 3.2 shows an example of two resource hierarchies and the type of resource instances that they contain.

Resource instances can be represented by the path name from their resource hierarchy root node. The function resource `main` can be represented by the path `/Code/main.C/main`. The path represents its relationship to other resources in the Code hierarchy; `main` is a **Function** resource whose parent is the **Module** resource `main.C`, and `main.C`'s parent is the Code hierarchy root node resource. The resource `/Code` represents all of the application's code, the resource `/Code/main.C` represents all code in module `main.C`, and the resource `/Code/main.C/main` represents all code in function `main` of module `main.C`.

Resource Class	Examples of type-specific information
Resource Base	name list of child resource instances (possibly empty) parent resource instance constraint method
Function Derived	address size source code line numbers
Module Derived	address size source code file name
Process Derived	process identifier address of stack pointer, heap, text & data segments /proc file descriptor
Semaphore Derived	address of semaphore variable type (binary or counting)
Machine Derived	internet address
File Derived	file descriptor

Figure 3.1 Example Resource Classes. *The Resource base class implements the characteristics that are common to all program resources, derived classes implement characteristics that are particular to a specific type of resource*

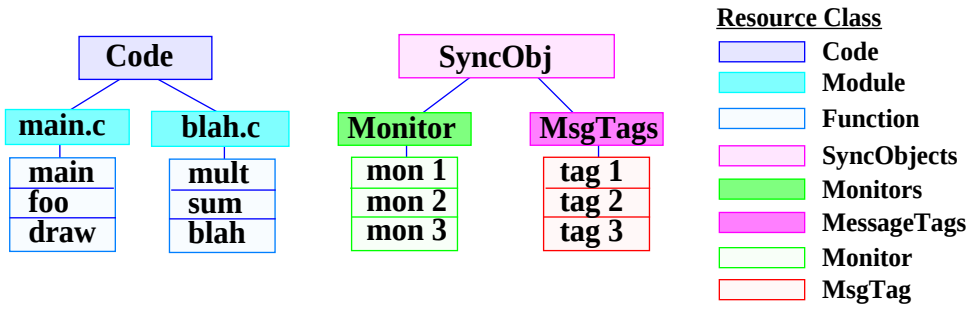

Figure 3.2 Example of Types of resource class instances in different resource hierarchies.

3.1.2 Representing the VM and AP Programs

The VM is a program that runs on an operating system/architecture platform. It might be represented as: **VM =** `[Machine,Process,Code,SyncObj]`, as shown in the left half of Figure 3.3.

The AP is input to the VM that runs it. Just like any other program resources, the VM's input can be represented by a resource hierarchy in the VM's program execution. However, the AP is a special form of program input; it is itself an executing program, and as such can be represented by a program execution (a set of resource hierarchies). For example, an interpreted application might

be represented by the following resource hierarchies: **AP =** `[Code,Thread,SyncObj]` (shown in the right half of Figure 3.3). AP's `Code` hierarchy represents a view of the AP as two **Class** resource instances (`foo.class` and `blah.class`), each consisting of several **Method** resource instances. The `SyncObj` hierarchy represents a view of the AP's synchronization objects, and the `Thread` hierarchy represents a view of the AP's threads. The resource hierarchies in Figure 3.3 represent the AP and VM executions separately.

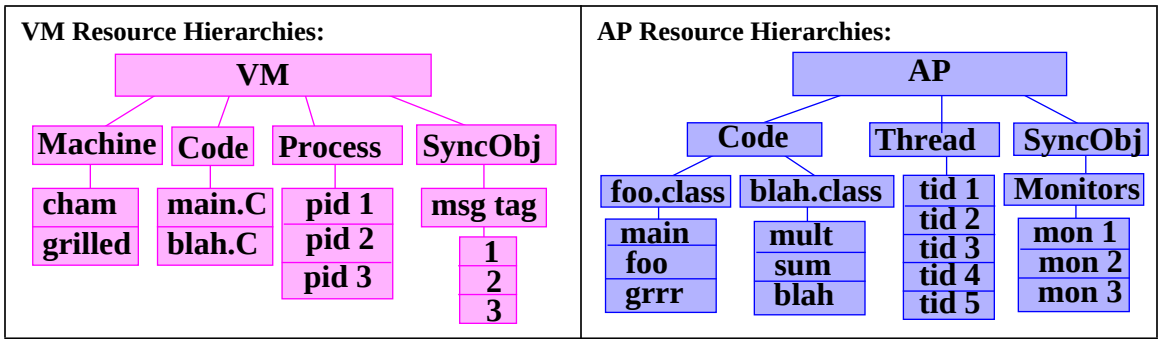

Figure 3.3 Example of resource hierarchies for the virtual machine and the application program.

3.1.3 Representing Interacting Programs

Both the VM's and the AP's executions are represented by a set of program resource hierarchies. However, there is an interaction between the execution of the two that must be represented by the program execution model. The relationship between the VM and its input (the AP) is that VM *runs* AP.

In an interpreted execution, the structure of the interpreted application program determines, in part, which VM code is executed, and the implementation of the VM determines how the AP is executed. We represent the relationship between the VM and the AP (the relationship "VM runs AP") by the union of VM and AP program resource hierarchies. The relationship between VM and AP resources is further described by resource constraints (Section 3.2) and resource mapping functions (Section 6.1.2). Figure 3.4 shows an example of resource hierarchies representing an interpreted execution using the example hierarchies from Figure 3.3. In this example, the execution of the AP by the VM is represented by the resource hierarchies: **"VM runs AP" =** `[Machine,VMCode,VMProcess,VMSyncObj,APThread,APSyncObj,APCode]`. Using the representation of "VM runs AP", we can select resources from VM and AP hierarchies to represent specific VM-AP interactions in the interpreted execution. We discuss the representation of specific VM-AP interactions in Section 3.2.

The *runs* relationship can be applied to more than two executing programs to describe a layered model. Since (A runs B) is a program execution, we can apply the runs relation to (A runs B), and another program execution C to get ((A runs B) runs C)) which is the program execution that represents the execution of C by B by A.

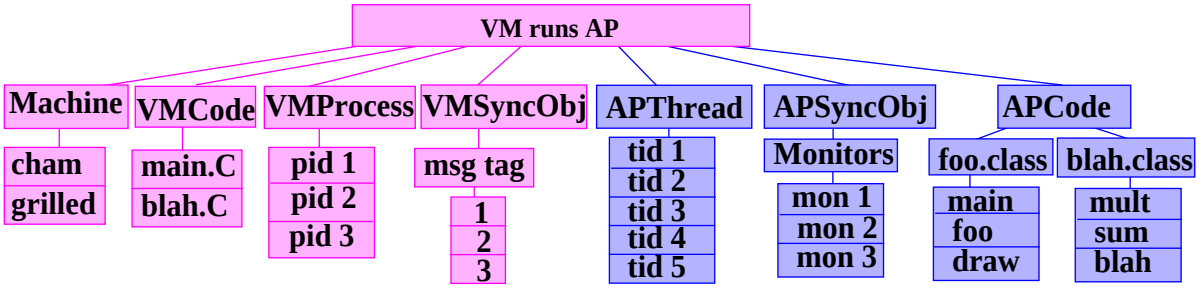

Figure 3.4 Resource hierarchies representing the interpreted execution.

3.2 Representing Constrained Parts of Program Executions

To query for and represent performance data that measure specific run-time activities, we need a way to specify constrained parts of the program's execution with which we can associate performance measures. We introduce the concept of a *focus* to represent constrained parts of a program execution; a focus uses *constraints* from each of its resource hierarchies to specify a specific part of the program execution. We first discuss constraints on individual resource hierarchies and then we define a focus.

3.2.1 Active Resources and Constraint Functions

A *constraint* on a resource hierarchy represents a restriction of the hierarchy to a sub-set of its resources.

- Definition 3.1: A *constraint, r,* is a single res csource instance from a resource hierarchy. It represents a restriction of the hierarchy to the subset of its resources represented by *r.*

For example, the constraint represented by the resource instance `/Process/pid_1` indicates that we are interested only in parts of the program execution that involve process `pid_1`. To associate performance measures only with the parts of the execution in which process `pid_1` is actively involved, a performance tool needs to identify the execution time intervals for which the constraint `/Process/pid_1` is *active.*

- Definition 3.2: Intuitively, a constraint, *r*, is *active* during only those parts of the program execution involving resource *r.* The precise definition of an active constraint is dependent on the constraint's resource type. For example, the constraint `/Code/main.c/main` is active when a program counter is in function `main`. A list of definitions for active is presented in Figure 3.5.

- Definition 3.3: A *constraint function, `constrain(r)`,* is a boolean, time-varying function of a constraint, *r*, that is true for the execution time intervals when *r* is active.

For example, Figure 3.7 shows the execution time interval when the constraint function of `/Code/main.C/msg_send` is true. Constraint functions are *resource-specific*; all constraint functions test whether their resource argument is active, however the type of test that is performed

Resource Type `example instance`	**Intuitive Definition of Active**	**Example Active Test**
Function `/Code/foo.c/foo`	Exclusive: A program counter (PC) is in Function `foo`'s code	Test if PC in function `foo`
	Caller: `foo` is the caller of the function in which the PC is	**Fuction** `foo` is in the caller location on a stack
	Inclusive: the function `foo` has been entered by not returned from	**Function** `foo` is on a stack
Module `/Code/foo.c`	At least one of module `foo.c`'s functions is active (can be exclusive, inclusive, or caller definition of active)	The OR of the active test for `foo.c`'s **Function** instances
Semaphore `/SyncObj/Semaphore/mutex`	A synchronization operation on semaphore `mutex` is active	`mutex` is an argument to the active semaphore `P` or `V` **Function** resource instance
Semaphores `/SyncObj/Semaphore`	A synchronization operation on a semaphore is active	The OR of the active test applied to **Semaphore** instances
MessageTag `/SyncObj/MsgTag/tag_1`	A send or receive is active for a message with message tag `tag_1`	`tag_1` is an argument to the active `send` or `receive` **Function** resource instance
MessageTags `/SyncObj/MsgTag`	A message send or receive is active	The OR of the active test applied to **MessageTag** instances
File `/Files/fid_1`	A file operation such as read, write, or seek on file `fid_1` is active	`fid_1` is an argument to the active `read`, `write`, or `seek` **Function** resource instance
Process `/Process/pid_1`	Process `pid_1` exists	Test for process `pid_1` in /proc file system
Machine `/Machine/cham`	Machine `cham` is involved in execution	Check if an application process has been created on `cham`

Figure 3.5 Active Definitions for instances of different Resource classes.

depends on the type of the resource argument; each derived resource class implements its own constraint method. Figure 3.5 lists example definitions of *active* for resource objects of various class types, and lists an example of how the test for *active* is implemented in a constraint method. For example, one definition of a **Function** resource being active is that the program counter is in the function. A constraint method to test for an active **Function** resource instance tests the program counter to see if it is in the function.

The test for a synchronization object being active is that a synchronization operation is occurring on that object. For example, the **Semaphore** resource `mutex` is active when it is the argument to an active semaphore function (`P` or `V`). Similarly, **MessageTag** resource `tag_1` is active when it is the argument to an active `send` or `receive` function, and **File** resource `fid_1` is active when it is the argument to an active file function (e.g., `read`, `write`, `seek`).

The test for a **Process** resource being active is a bit strange; it tests whether the process exists. Since the exact execution state of a process (such as run-able or blocked) is invisible, the only definition for an active process is that the process exists.

Some types of resources can have more than one definition of *active*. For example, a **Function** resource can be active if a program counter is in the function, or it can be active if it is on the execution stack, or it can be active if it is the caller of the function in which a program counter is. These are three different ways to constrain a **Function** resource instance, therefore we need three *situation-specific* constraint methods for the **Function** class. Each situation-specific constraint method implements a different test for an *active* function to distinguish between the exclusive, inclusive, and caller definition of active.

Constraining a resource is something that is common to all resource types. Therefore the **Resource** base class exports a constraint method (`constrain`). Since the particular test for *active* depends on the resource's type, each derived resource class overrides the base class' `constrain` method with its own resource-specific implementation. Also, derived resource classes implement any situation-specific constraint methods. We differentiate between a class' situation-specific methods by name. For example, the **Code**, **Module**, and **Function** classes implement multiple constraint methods, one for each way in which a code object can be constrained: `constrain` is the exclusive constraint method, `constrain_inclusive` is the inclusive constraint method, and `constrain_caller` is the caller constraint method. Not all resource classes have multiple constraint functions.

Calls to a constraint method reference the resource object as a path name relative to that object minus the hierarchy resource name. For example, a call to `Code.constrain(main.c/main)` will result in a call to the `constrain` method of the **Function** class for the resource `/Code/main.c/main`. A class' constraint method(s) are implemented to use their path name argument to traverse the resource hierarchy and find the correct resource instance. Once the resource instance has been found, its `constrain` method is invoked. Figure 3.6 shows the algorithm for implementing a class' constraint method(s).

3.2.2 Constraint Operators

Constraint methods can be combined by constraint operators to create more complicated boolean expressions that describe when constrained parts of the program execution are active.

- Definition 3.4: The set of constraint function operators are AND, OR.
  - **AND:** (constrain1(r_i)) **AND** (constrain2(r_j)):
    both constrain1(r_i) and constrain2(r_j) are true.
  - **OR:** (constrain1(r_i) **OR** constrain2(r_j)):
    either constrain1(r_i) or constrain2(r_j) or both are true.

```
bool constrain(path_name) {

   // if we are at the right level, apply the active test
   if (path_name == null)
         return active();

   // if we are not at the right level, call the child's constrain method
   child = find_child_resource(head(path_name));
   if (!child) { return false;}
   return(child.constrain(tail(path_name));
}
```

Figure 3.6 Generic algorithm for implementing a Resource Class' constrain method. *A* `constrain` *method traverses the hierarchy using its* `path_name` *argument. If there are still resources in the path name, it finds its child resource corresponding to the next level in the path, and calls its child's* `constrain` *method. If it is at the level corresponding to the end of the path, then it applies its constraint test to the resource.*

Constraints combined with constraint operators are true for certain time intervals of the program execution; they are true for the parts of the program that the combination of their constraints represent. For example, the expression `(Code.constrain(libc.so/read) AND File.constrain(fd2))` is true over execution time intervals when a read on file descriptor `fd2` is active. Figure 3.7 shows examples of constraint functions combined with different constraint operators and the resulting execution time intervals when each tests true.

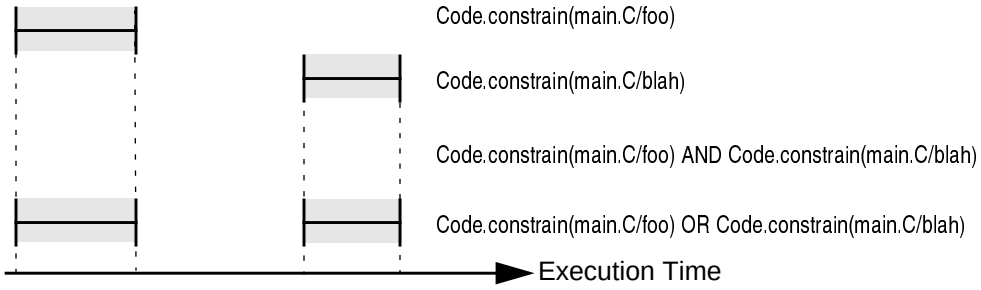

Figure 3.7 Constraint tests for constraints combined with constraint operators. *The diagram shows the time intervals for two events (when functions* `foo` *and* `blah` *are active), and it shows the time intervals when these events combined with constraint operators are true. In this example, the AND of the two events never tests true because* `foo` *and* `blah` *are not active over overlapping execution time intervals.*

When constraint operators are used to combine constraints from programs with multiple threads or processes, care must be taken to correctly combine the constraints in the presence of simultaneously running threads or processes. For example, **Process** `pid_1` executing the `P` or `V` method on **Semaphore** `mutex` is represented by `Process.constrain(pid_1) AND SyncObj.constrain(mutex)`. If **Process** `pid_2` is executing `P` or `V` on **Semaphore** `mutex` over

execution time intervals that overlap with the time intervals when **Process** `pid_1` is active but `pid_1` is not executing `P` or `V` on **Semaphore** `mutex`, then `Process.constrain(pid_1) AND SyncObj.constrain(mutex)` should evaluate to false even though the execution time intervals overlap when the constraints are active. To solve this problem, `constrain` methods from programs with multiple threads or processes return a vector of true and false values (one for the `constrain` method applied to each thread), and the constraint operators combine individual elements in these vectors.

To prevent an incorrect evaluation of the constraint operator in the presence of simultaneously running threads, a constraint operator is applied to corresponding individual elements from constraint's vector of boolean values to produce a result vector. The values in the result vector are combined with OR to obtain the final result. The following is the algorithm for combining N constraints with a constraint operator:

1. Each `constrain` method is evaluated to produce N vectors of boolean values.
2. The constraint operator is applied to corresponding elements from each of the N vectors to produce a result vector (i.e. the i^{th} element from one vector is combined with the i^{th} element from the other vectors to produce the i^{th} element of the result vector).
3. The individual boolean values in the result vector are combined with OR to obtain the final true or false result.

Examples 1 and 2 from Figure 3.8 show `constrain` methods combined with AND. The first example evaluates to true because the first element from each `constrain` method's vector is true, and the second example evaluates to false because the AND of each corresponding vector element is false. The third example shows the constraints from the second example combined with the OR operator. In this case the final result is true.

```
1. SyncObj.constrain(mutex)        [T, F ... , F]
   AND                              &  & ...   &
   Process.constrain(pid_1)        [T, F ... , F]
                                =  [T, F ... , F] = TRUE

2. SyncObj.constrain(mutex)        [T, F ... , F]
   AND                              &  & ...   &
   Process.constrain(pid_2)        [F, T ... , F]
                                =  [F, F ... , F] = FALSE

3. SyncObj.constrain(mutex)        [T, F ... , F ]
   OR                              or or ...  or
   Process.constrain(pid_2)        [F, T ... , F]
                                =  [T, T ... , F] = TRUE
```

Figure 3.8 An example of applying constraint operators for programs with multiple threads. *Each constrain method returns a vector of true or false values. Next, each corresponding element of the two vectors is combined with the constraint operator to produce a result vector, then the elements in the result vector are combined with OR to obtain the final true or false result.*

3.2.3 Properties of Constraint Operators

Multiple constraints can be combined with multiple constraint operators to represent more complicated constrained parts of the execution of "VM runs AP", and to represent performance data for those events. We combine constraints with AND and OR as they are normally defined, and thus all properties of the standard definition of AND and OR hold (idempotency, commutativity, associativity, absorption and distributivity). Figure 3.9 lists the properties of constraint operators and shows how constraint operators are combined. These properties are used to determine how to combine multiple constraints. In Section 3.3.4, we discuss combining constraints in our representation of performance data.

Property	AND	OR
Idempotency	`(A AND A) = A`	`(A OR A) = A`
Commutativity	`(A AND B) = (B AND A)`	`(A OR B) = (B OR A)`
Associativity	`(A AND B) AND C) =` `(A AND (B AND C))`	`(A OR B) OR C) =` `(A OR (B OR C))`
Absorption	`(A OR (A AND B)) = A`	`(A AND (A OR B)) = A`
Distributivity	`(A AND (B OR C)) =` `(A AND B) OR (A AND C)`	`(A OR (B AND C)) =` `(A OR B) AND A OR C)`

Figure 3.9 Properties of Constraint Operators.

3.2.4 Foci

By combining one constraint from each resource hierarchy, specific parts of the running program are represented. We call this representation a *focus*.

- Definition 3.5: A *focus* is a selection of resources, one from each resource hierarchy. It represents a constrained part of the program's execution. A focus is active when all of its resources are active (it is the AND of the constraint functions, one constraint function from each resource hierarchy).

An example focus is `</Threads/tid2,/SyncObj/Monitors/mon_1, /Code/foo.class/foo>` from the AP resource hierarchies shown in Figure 3.3. It represents the part of the AP's execution when thread `tid2`, monitor `mon_1`, and method `foo` are active. The part of the execution represented by this focus is occurring when the AND of its corresponding constraint functions is true: `Threads.constrain(tid2) AND SyncObj.constrain(Monitor/mon_1) AND Code.constrain(foo.class/foo)`.

Constraining a resource hierarchy to its root node is the same as not constraining the hierarchy. For example, the focus `</Process,/Code/main.c>` is active when `/Code/main.c AND`

/Process are active, which is the same as when /Code/main.c is active; constraining to all processes is equivalent to not constraining to a particular process.

3.3 Representing Performance Data from Interpreted Executions

To describe performance data that measure the interaction between an AP and a VM, we provide a mechanism to selectively constrain performance data to any aspect of the execution of "VM runs AP". There are two complementary (and occasionally overlapping) ways to do this. One is to create functions that measure specific aspects of performance, and the other is to restrict performance measurement to foci that represent restricted parts of the execution.

3.3.1 Using Foci to Constrain Performance Data

One way to selectively constrain performance data is to associate a performance measure with a focus from "VM runs AP" that describes a constrained part of the execution of the application program by the virtual machine. If the focus contains resources that are constrained on both VM and AP resource hierarchies, then it represents a specific VM-AP interaction. The following is an example from Figure 3.4: </Machine,/Code/main.C/invokeMethod,/Process/pid_1,/SyncObj,/APThreads,/APSyncOjbect,/APCode/foo.class/foo>, represents the part of the VM's execution when process pid_1 is executing function invokeMethod in main.C, and is interpreting instructions in method foo of the AP. By combining metrics with this focus, performance data for the specified VM-AP interaction can be described.

3.3.2 Using Metrics to Constrain Performance Data

The second way to selectively constrain performance data is by specifying a metric that measures a particular aspect of performance. For example, CPU utilization, synchronization waiting time, and number of procedure calls are performance metrics.

- Definition 3.6: A *metric function, f,* is a time-varying function that consists of two parts (*f* = [C] T):
  (1) Constraint functions [C] : an expression of constraint functions combined with constraint operators.
  (2) Counter or timer function T: a function that counts the frequency or time of program events. Examples include process time, elapsed time and procedure calls.
  A metric function's counter or timer function, T, accumulates value only during those execution time intervals when the metric function's constraint functions, [C], are true.

Metric functions are created by measuring their timer or counter functions only during the execution time intervals when their constraint functions are active. Some metric functions have null constraints. For example, the following are metric functions with null constraints:

- **CPU_utilization = [ ]processTime /sec**. Amount of process time per second.
- **execution_time = [ ]wallTime /sec**. The amount of wall time per second.

More complicated metric functions can be built by combining timer or counter functions with constraints, as in the following examples:

- **io_wait = [Code.constrain(libc.so.1/read) OR Code.constrain(libc.so.1/write)] wallTime/ sec**.
  The amount of time spent reading plus the amount of time spent writing.

- **io_count = [Code.constrain(libc.so.1/read) OR Code.constrain(libc.so.1/write)] count/sec**.
  The number of times per second that either read or write are executed.

- **func_calls = [Code.constrain(*/*)] count/sec.**
  The number of functions called per second.

The metric function `func_calls` contains a constraint using asterisk symbols to indicate that the constraint should be applied to all resources at that level in the hierarchy. For example, `Code.constrain(*/*)` means apply the `constrain` method to all **Function** resource instances of all **Module** resource instances; the metric's count function should be incremented each time a function becomes active in the program execution.

A *metric* consists of a metric function and a specification of how to apply the metric function to different foci. We will discuss combining metrics with foci in more detail in Section 3.3.4.

3.3.3 Metric Functions for Interpreted Executions

VM-specific metric functions measure activities that are specific to a particular virtual machine. VM-specific metrics are designed to present performance data to an AP developer who may have little or no knowledge of the internal workings of the VM; they encode knowledge of the virtual machine in a representation that is closer to the semantics of the AP language. Thus, an AP developer can measure VM costs associated with the execution of the AP without having to know the details of the implementation of the VM. For an interpreter that fetches and decodes an AP code object on each execution, we could define a metric **timeToFetch** to measure the time to fetch a code object, and a metric **timeToDecode** to measure the time to decode a code object. These are examples of VM-specific metric functions that measure a particular interaction between the AP and the VM (the VM fetching and decoding AP code). The knowledge of the parts of the VM that perform these actions are encoded in the VM-specific metric. In other words, the VM-specific metric function contains constraints on VM resource hierarchies. For example, if the VM routine `fetchCode()` gets the next set of AP instructions to interpret, then **timeToFetch** may be defined as: `[VMCode.constrain(main.C/fetchCode)] cpuTime/sec`.

3.3.4 Combining Metrics with Foci from VM runs AP

We represent performance data as a metric–focus pair. A performance metric consist of a metric function and a specification of how to apply the metric to different foci (which constraint functions to apply to each of the resource hierarchies).

- Definition 3.7: A *metric* is a measure of some aspect of performance of a program execution. It consists of two parts: (1) a metric function, and (2) a specification of how to apply the metric to different foci.

We specify how to apply the metric to different foci by indicating which constraint method to apply to the foci's resources; we list the name of the resource-specific or situation-specific constraint method from each resource hierarchy. It does not always make sense to apply a metric to any possible focus from the program execution. For example, it does not make sense to associate the `func_calls` metric with a focus constrained to a particular message tag. For the `func_calls` metric, we do not specify a constraint function from the `SyncObj` resource hierarchy. As a result, the specification of how to apply the metric to different foci also indicates on which hierarchies a focus can be constrained. Figure 3.10 shows examples of metric definitions.

Metric	Metric Function	How to apply to foci

`func_calls`	`[Code.constrain(*/*)] count/sec`	`Code.constrain`
		`Process.constrain`
		`Machine.constrain`

`cpu`	`[]processTime/sec`	`Code.constrain`
		`SyncObj.constrain`
		`Process.constrain`
		`Machine.constrain`

`cpu_inclusive`	`[]processTime/sec`	`Code.constrain_inclusive`
		`SyncObj.constrain`
		`Process.constrain`
		`Machine.constrain`

`io_wait`	`[Code.constrain(libc.so/read) OR Code.constrain(libc.so/write)] wallTime/sec`	`Code.constrain_caller`
		`SyncObj.constrain`
		`Process.constrain`
		`Machine.constrain`

Figure 3.10 Example Metric Definitions. *Metric function plus how to apply the metric to a given focus.*

For example, the **io_wait** metric, when combined with a focus refined on the `Code` hierarchy, uses the **Code** class' `constrain_caller` situation-specific constraint method to attribute the reading and writing time to the function that called `read` or `write`. For the CPU metrics, any focus refined on the `Code` hierarchy will use the `constrain` method to compute CPU exclusive time (**cpu**), and the `constrain_inclusive` method to compute CPU inclusive time (**cpu_inclusive**). Neither CPU metric can be combined with foci constrained on the `SyncObj` hierarchy.

Metrics combined with foci restrict the metric to a specific part of the program execution. The result is the AND of the metric and the focus. For example:

- **(CPU, < /Process/pid1, /Code/main.c/foo>)**
  process time when process `pid1` is executing code in function `foo`:
  `= ([Process.constrain(pid1)] AND`

```
    [Code.constrain(main.c/foo)])
  AND [](processTime/sec)
```

- **(io_wait, </Code/main.c/foo>)**
  I/O blocking time due to I/O calls from function `foo`:

```
=  Code.constrain(main.c/foo) AND
   ([Code.constrain(libc.so.1/read)
     OR Code.constrain(libc.so.1/write)] wallTime/sec)
=   ([Code.constrain(main.c/foo) AND
       Code.constrain(libc.so.1/read)]
     OR [Code.constrain_caller(main.c/foo) AND
       Code.constrain(libc.so.1/write)]) (wallTime/sec)
```

Combining a metric value with a focus from "VM runs AP" constrains performance measurement to a specific event in the execution of the application program by the virtual machine. The following are some examples of performance data represented as metric-focus pairs from "VM runs AP":

1. **(CPUtime, </VMCode/main.C/invokeMethod>)**: Process time spent in VM function `invokeMethod`.
2. **(CPUtime, </APCode/foo.class/foo>)**: Process time spent in AP function `foo`.
3. **(numCycles, </APCode/foo.class/foo>)**: Number of cycles spent in AP function `foo`. This type of measurement may be possible for a simulator VM running an AP where the simulator keeps its own representation of time (in this case a cycle count associated with executing AP code), or it may be possible on machines with hardware cycle counters that are accessible to the performance tool.
4. **(CPUtime, </VMCode/main.C/invokeMethod, /APCode/foo.class/foo>)**: Process time spent in the VM function `invokeMethod` while the VM is interpreting AP function `foo`. If `invokeMethod` is called to set up state or to find the target of a call instruction each time a method call instruction is interpreted in method `foo` code, then this metric-focus combination measures the method call overhead associated with all method calls made by AP function `foo`.
5. **(methodCallTime, <APCode/foo.class/foo>)**: Same performance measurement as example 4, except it is represented by a VM-specific metric `methodCallTime` combined with a focus that is refined on only the AP resource hierarchies (this performance data is in terms of an AP-developer's view of the interpreted execution.

Examples 4 and 5 above illustrate how a VM-specific metric function, combined with an AP focus, is equivalent to another metric function combined with a focus containing both AP and VM resources. The **methodCallTime** metric encodes information about VM function `invokeMethod` that is used to compute its value. In general, there is a relationship between some metrics and foci; a metric implies part of a focus: **methodCallTime** = `[VMCode.constrain(main.C/invokeMethod)] CPUtime/sec`. Computing the **methodCallTime** metric involves identifying and measuring locations in the VM that perform the activity that the metric is measuring; when **methodCallTime** is applied to a focus, `invokeMethod` is an implied part of the focus;

Examples 4 and 5 also demonstrate how the same performance measure can be represented in terms of both the VM developer's and the AP developer's view of the execution. Example 5 is performance data that is in a language that an AP developer can understand. Typically, the AP developer has little or no knowledge of the internal structure of the VM. Therefore, the AP developer is not able to specify specific VM-AP interactions by specifying a focus refined on both AP and VM resource hierarchies. Instead, the AP developer combines a VM-specific metric with a focus that is refined on only AP resource hierarchies. The VM-specific metric encodes information about the virtual machine in a form that represents the AP developer's view of how the AP is executed by the VM.

On the other hand, the VM developer knows the internals of the VM and wants to see platform specific costs associated with constrained parts of VM's execution characterized in terms of the parts of the AP that it runs. For a VM developer, performance data associated with specific VM-AP interactions is represented as a focus that is constrained on both VM and AP hierarchies combined with a non-VM-specific metric.

3.3.5 Performance Data Associated with Asynchronous Events

There are some types of activities that cannot be easily represented by foci. Foci represent an active state of the interpreted execution. For asynchronous activities like garbage collection and writes, we may want to see performance data associated with this activity and with the part of the program's execution that triggered this activity. For example, we may want to see the amount of time spent garbage collecting a particular object associated with the function in which the object was freed. Since garbage collection is asynchronous, the function that frees the object is not necessarily active when the object is garbage collected.

We represent performance data that measures asynchronous activities as a special type of metric (an asynchronous metric) associated with a focus. Asynchronous metric functions consist of two parts: one part is a function for setting state when the focus is true, the other part is a function for computing the performance measure when the state is true. For example, the metric function **async_GTime,** which measures garbage collection time associated with an object and the function that freed the object, consists of the following two parts: (1) when the object goes out of scope in the function, update a state table with the object's identifier, and the function's identifier; (2) when an object is garbage collected, if it has a state table entry, then start a timer and stop a timer around its GC activity and assign this cost to the function.

In general, to compute asynchronous metrics some extra state must be kept that specifies which part of the program to attribute an activity. Purify [24] is an example of a debugging tool that does something like this to identify memory access errors at run-time and to associate these with parts of the application code where the memory was accessed, allocated, initialized and/or de-allocated.

3.4 Conclusions

In this chapter we presented a representational model for describing performance data from interpreted executions that can explicitly represent VM-AP interactions, and that is in a form that both the VM and AP developers can understand. Our model allows an AP developer to see inside the VM an understand the fundamental costs associated with the VM's execution of the AP. It also allows a VM developer to characterize the VM's performance in terms of the AP code that it executes. Our model is a guide for what to build into a performance tool for measuring interpreted program executions.

Chapter 4

Paradyn-J: A Performance Tool for Measuring Interpreted Java Executions

We present Paradyn-J, a performance tool for measuring interpreted Java executions. Paradyn-J is a proof-of-concept implementation of our model for representing performance data that describes interactions between the VM and the AP. We discuss Paradyn-J's implementation, and demonstrate how performance data from Paradyn-J can answer questions about the Java application's and the Java virtual machine's execution. We present results from a performance tuning study of a Java application program; using performance data from Paradyn-J, we tune a Java application and improve its performance by more than a factor of 1.5. In Chapter 7 we discuss alternative ways in which a Java profiling tool based on our model can be implemented.

4.1 Paradyn-J's Implementation

Paradyn-J is an extension of the Paradyn Parallel Performance Tools [47] for measuring interpreted Java executions. Paradyn-J is implemented for versions 1.0.2 and 1.1.6 of Sun Microsystem's JDK running on SPARC-Solaris platforms.

We first discuss those parts of the Java VM with which our tool must interact to measure interpreted Java executions. We then discuss Paradyn-J's implementation. In particular, we discuss how Paradyn-J discovers AP program resources, how Paradyn-J instruments Java AP byte-codes and Java VM native code to collect performance data that measure specific VM-AP interactions, and how Paradyn-J provides performance data in terms of both the VM developer's and the AP developer's view of the interpreted execution.

4.1.1 The Java Virtual Machine

We briefly discuss the Java virtual machine and how it executes Java application byte-codes. The Java Virtual Machine [41] is an abstract stack-based processor architecture. A Java program

is written in the Java object-oriented programing language. A Java program consists of some number of classes. Each class is compiled into its own .class file by the Java compiler. A class' method functions are compiled into byte-code instructions. The byte-code instructions are the Java virtual machine's instruction set; Java byte-codes can run on any platform that has a Java VM implementation.

The Java VM starts execution by invoking the `main` routine of the Java application class. To execute `main`, the VM loads the .class file containing `main`, verifies that the .class file is well-formed, and initializes the class (initializing the superclass of the class first). The VM delays binding each symbolic reference (i.e. method call target) from the class until the initial time each reference is executed. By delaying binding, the VM only incurs loading, verification, and initialization costs for classes that are necessary to a particular execution of the program. As a result, Java application class files can be loaded at any point in the interpreted execution.

The VM divides memory into three parts: the object memory, the execution stack, and the class code and constantpool. The *class code and constantpool* part contains the application's method byte-codes and the per-class constantpools. A class' *constantpool* is like its symbol table. The object memory is used to store dynamic objects in the program execution (object instances, local variables, method arguments, and method operand stacks); associated with each method is an operand stack, and the executing byte-code instructions push and pop operand values from this stack. The final area of memory is the execution stack. There is one execution stack for each active thread; the execution stack consists of a number of frames. Each time a method is called a new stack frame is created to hold the state of the executing method. When a method call returns, its frame is popped off the stack and execution continues in the calling frame.

The virtual machine contains four registers used to hold the current state of the machine; **PC** points to the next byte-code instruction to execute, **Frame** points to the current stack frame, **opTop** points to the top of the executing method's operand stack, and **Vars** points to the method's local variables. At any point in the execution, the Java VM is executing code from a single method[1]. The registers and stack frame are used to locate all state necessary to execute the method's byte codes. When a method call byte-code instruction is executed, the VM creates a new execution stack frame for the method, and allocates space in Object Memory for the method's operand stack, arguments, and local variables. Figure 4.1 shows the three areas of memory in the virtual machine.

Paradyn-J interacts with the Java VM routines that load Java application .class files, and accesses the VM's internal data structures that store AP method byte-codes, AP class constant-

1. For version 1.1.6 of JDK using Solaris threads and running on a multi-processor, it is possible that more than one thread is running at one time. In this case, the **Frame** register points to a list of execution stacks. Each element in the list contains an execution stack for one of the currently running threads.

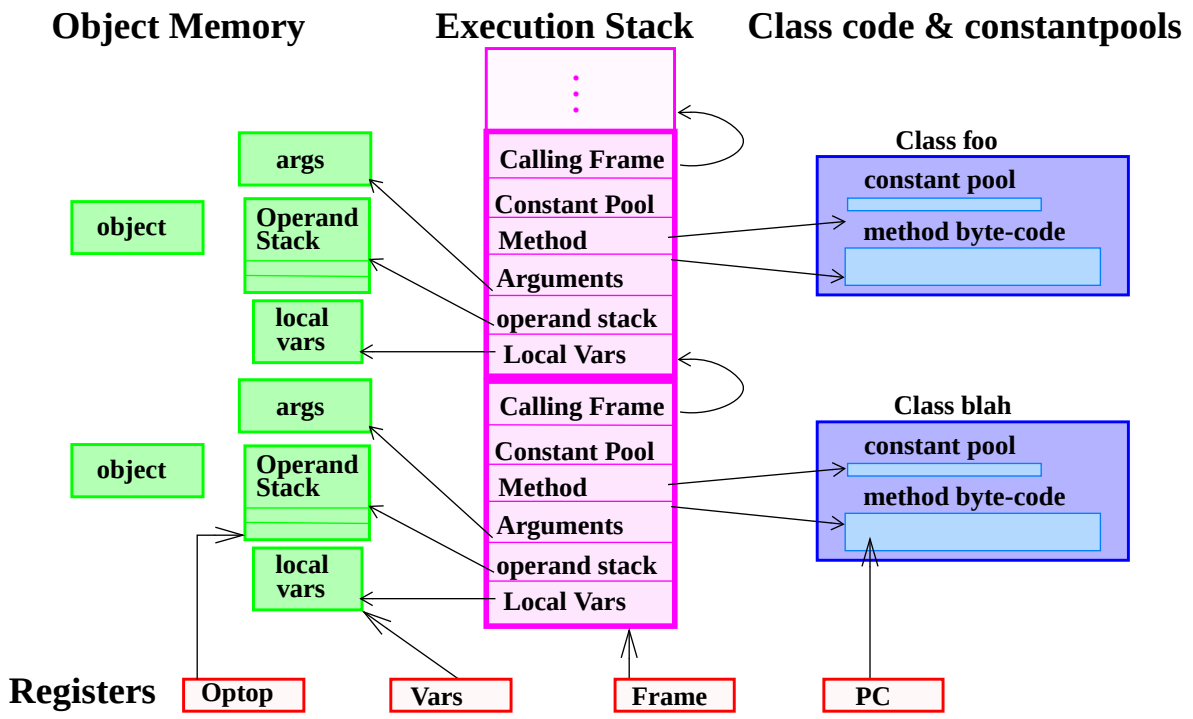

Figure 4.1 Memory Areas of the Java Virtual Machine. *There are three main areas the class constant pool, the execution stack, and object memory.The VM also has four registers to define the current state of the system. The currently executing method is in the stack frame indicated by the* ***Frame*** *register.*

pools, execution stack frames, VM registers, and application's operand stacks to instrument Java AP byte-codes

4.1.2 Parsing Java .class Files and Method Byte-codes

New class files can be loaded by the Java VM at any point in the Java execution. Classes and their methods are program resources that Paradyn-J needs to discover and possibly measure; Paradyn-J must be able to discover new code resources whenever the VM loads an AP class file. To do this, instrumentation code is added to the Java VM routines that perform class file loading. When a class file is loaded by the VM, instrumentation code is executed that passes Paradyn-J the information necessary to locate the loaded class.

Part of class file loading includes the VM transforming the class into an expanded form that is easier to execute. We delay parsing the class and its method byte-codes until after the class has been transformed by the VM. We instrument the functions in the VM that perform these transformations to find the location of the transformed method byte-codes. We then parse the transformed byte-codes and create method resources for the class, and we find the method's instrumentation points (the method's entry, exit, and call sites). At this point, instrumentation requests can be

made for the class' methods. Measurement of the loading time is initiated before the class' resources are created.

4.1.3 Dynamic Instrumentation for VM Code

Paradyn-J uses Paradyn's dynamic instrumentation [27] to insert and delete instrumentation code into Java virtual machine code at any point in the interpreted execution. Paradyn's method for instrumenting functions is to allocate heap space in the application process, generate instrumentation code in the heap, insert a branch instruction from the instrumented function to the instrumentation code, and relocate the function's instructions that were replaced by the branch to the instrumentation code in the heap. The relocated instructions can be executed before or after the instrumentation code. When the instrumented function is executed it will branch to the instrumentation code, execute the instrumentation code before and/or after executing the function's relocated instruction(s), and then branch back to the function.

Because the SPARC instruction set has instructions to save and restore stack frames, the instrumentation code and the relocated instructions can execute in their own stack frames and using their own register window. This way instrumentation code will not destroy the values in the function's stack frame or registers. Figure 4.2 shows an example of dynamically instrumenting a VM function.

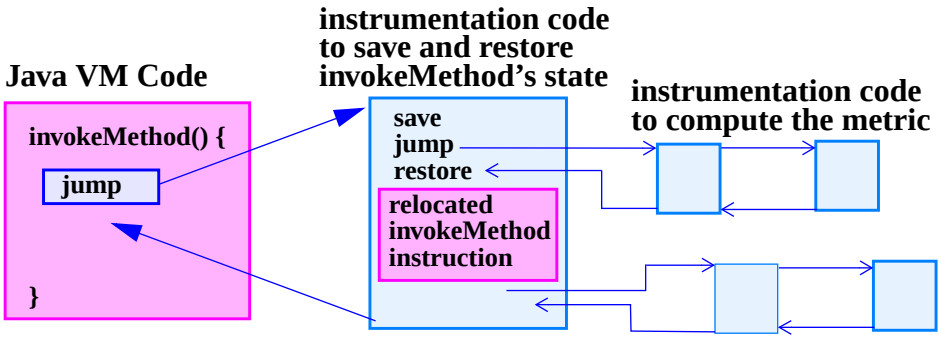

Figure 4.2 Dynamic Instrumentation for Java VM code. *In this example VM function* `invokeMethod` *is instrumented. An instruction in* `invokeMethod` *is replaced with a branch instruction that jumps to the instrumentation code in the heap, and the overwritten* `invokeMethod` *instruction is relocated to the instrumentation code.*

4.1.4 Transformational Instrumentation for AP Code

We use an instrumentation technique called *Transformational Instrumentation* to dynamically instrument Java application byte-codes. Our technique solves two problems associated with instrumenting Java byte-codes at run-time. One problem is that there are no Java Class Library methods or JDK API's (prior to release 1.2) for obtaining CPU time for AP processes or threads. As a result, Paradyn-J must use some native code to obtain CPU time measures for instrumented

byte-codes. The second problem is that our byte-code instrumentation needs operand stack space, and argument and local variable space to execute. For every AP method on the call stack, the VM creates an execution stack frame, an operand stack, and argument and local variable space for executing the method's byte-code instructions; our instrumentation byte-codes also need this space to execute.

There are two ways to obtain extra execution space for our byte-code instrumentation. The first way is to allocate extra space on the instrumented method's operand stack and allocate extra argument and local variable space for the instrumentation code. When the VM executes the instrumentation code, the instrumentation instructions will execute in the method's stack frame using this extra allocated space. The second way is to insert a method call byte-code instruction to jump to byte-code instrumentation; the call will create a new execution stack frame, operand stack, and argument and local variable space for the instrumentation byte-codes.

If we use the first approach, and execute instrumentation byte-codes within the method's stack frame, then we must increase each method's maximum stack size, argument size, and local variable size when its .class file is loaded by the VM. We also need to change the class' constant-pool at load time to add entries to point to methods that our instrumentation code may call (method call byte-code instructions are resolved using constantpool entries). Changes to a method's operand stack, argument and local variable sizes must be done at the time the .class file is loaded. The load-time changes are necessary because at instrumentation time (when the VM is paused at any point in its execution to insert byte-code instrumentation), the VM may be updating the AP's current execution state. As a result, values used to represent AP's execution state may be in VM local variables on the stack or in machine registers; we cannot safely change any of AP's execution state at instrumentation time, because our changes may be overwritten by VM routines. Instead, every AP method's maximum stack size, argument size, local variable size, has to be modified when the .class file is loaded by the VM; an estimate of the maximum amount of space used by byte-code instrumentation must be added to these sizes. For similar reasons, we cannot modify the class' constantpool at instrumentation time. Therefore, entries for all possible methods called by instrumentation code must be added to the constantpool when the .class file is loaded. In most cases, many of the constantpool entries, and much of the extra space allocated on the operand stack and argument and local variable space will go unused, resulting in increased garbage collection and Java heap allocation for unused space.

The second approach to safely executing instrumentation code is to use a call instruction to jump to instrumentation code. When the VM interprets a call instruction, it creates a new execution context for the called method, so instrumentation code will execute in its own stack frame with its own operand stack. There are two problems with this approach. First, method instructions that are overwritten with calls to instrumentation code cannot be relocated to the instrumentation code in the heap; in the instrumentation code there is no way to restore the method's execution context that is necessary to execute the relocated byte-code instructions. Second, inter-

preting method call instructions is expensive. We solve the first problem by relocating the entire method to the heap with extra space for inserting the method call instructions that call instrumentation code. However, the second problem is unavoidable since method call instructions are already necessary for obtaining CPU measures from native code.

In Paradyn-J we use the second approach. We call our technique of relocating methods when first instrumented, and instrumenting byte-codes with native code is called *Transformational Instrumentation*. Transformational Instrumentation works as follows (illustrated in Figure 4.3): the first time an instrumentation request is made for a method, relocate the method to the heap and expand its size by adding `nop` byte-code instructions around each instrumentation point. When a branch to instrumentation code is inserted in the method, it replaces the `nop` instructions; no method byte-codes are overwritten and, as a result, all method byte-codes are executed using the their own operand stack and stack frame. The first bytes in the original method are overwritten with a `goto_w` byte-code instruction that branches to the relocated method. SPARC instrumentation code is generated in the heap, and method call byte-code instructions are inserted at the instrumentation points (the `nop` byte-code instructions) to jump to the SPARC instrumentation code.

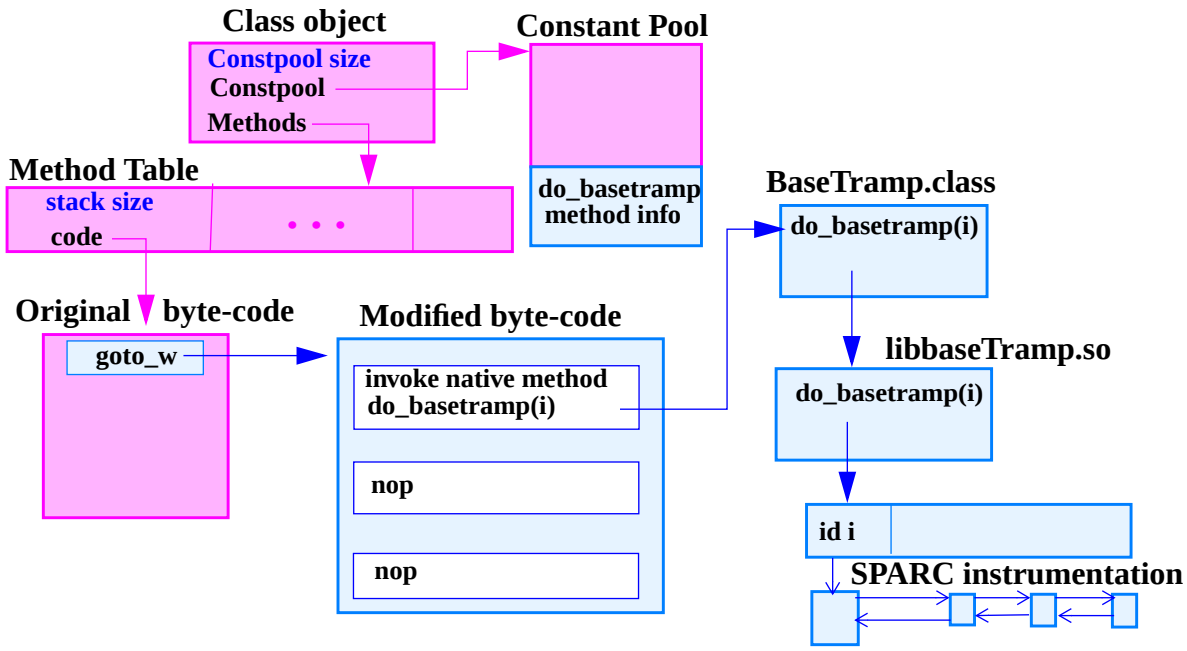

Figure 4.3 Transformational Instrumentation for Java application byte-codes. *The darker colored boxes represent pre-instrumented Java VM data structures, the lighter colored boxes are added to instrument a Java Method.*

The VM will create a new execution stack frame and operand stack for the instrumentation code, if the jump to the instrumentation code is made by a method call byte-code instruction; all branches to instrumentation code are calls to the static method `do_baseTramp(int id)` that

executes the instrumentation code. The `id` argument is used to indicate from which instrumentation point it was called. To call `do_baseTramp` we insert one byte-code instruction to push the `id` operand on the operand stack and one byte-code instruction to call the method. Also, since method calls are resolved using the calling class' constantpool, the class' constantpool must be modified to add entries that provide information about `do_baseTramp`. A class' constantpool only has to be modified once (when the class file is loaded), and only has to be modified with entries to resolve one method call (the call to `do_baseTramp`). Finally, the state of the virtual machine (its execution stacks, and register values) must be checked to see if it is safe to insert these changes. If it is not safe, the changes must be delayed until some point in the execution when it is safe to insert these changes. To implement delayed instrumentation, special instrumentation code is inserted in a method that is lower in the execution stack and that is at a point where Paradyn-J determines that it will be safe to insert the instrumentation. When this special instrumentation is executed it notifies Paradyn-J that it should attempt to insert any delayed instrumentation code.

We re-use much of Paradyn's code generation facilities for generating SPARC instrumentation code to generate instrumentation code for Java AP methods. SPARC instrumentation code can be used to instrument Java byte-codes if we define `do_baseTramp` to be a native method. Java's native method facility is a mechanism through which routines written in C or C++ can be called by Java byte-codes. The C code for `do_baseTramp` is complied into a shared library, and a Java `BaseTramp` class is created that declares `do_baseTramp` to be a static native method function. When this class is compiled, the Java compiler generates byte-code stub procedures for the native method that can be called by other byte-code to trigger the VM to load and execute the native code in the shared library.

The C implementation of the `do_baseTramp` routine contains a vector of function pointers that call SPARC instrumentation code. The `id` argument is an index into the vector to call the instrumentation code. The `do_baseTramp` method will return to the calling method via the native method interface.

To get the Java VM to execute the `do_baseTramp` method, first Paradyn-J has to get the VM to load the `BaseTramp` class file. One way to do this is to add instrumentation to the VM that will call its load routines to explicitly load the `BaseTramp` class. Another alternative is to find the main method and, before it is executed, instrument it to include a call to a BaseTramp method function. This will trigger the VM to load the `BaseTramp` class at the point when the function is called. The first option is better because Paradyn-J has control over when the `BaseTramp` class has been loaded and, as a result, knows when byte-code instrumentation can be inserted. In our current implementation, we get the application to load the `BaseTramp` class by modifying the application's source code; a call to a `BaseTramp` method is added to the application's `main` method. As a result, we have to re-compile one AP class (the source file that contains the method `main`). Paradyn-J can be implemented to get rid of this extra compiling step; the current version is a simplification for our prototype implementation.

Instrumentation type tags associated with AP and VM resources are used to determine if generated SPARC instrumentation code should be inserted into Java method byte-codes using Transformational Instrumentation or should be inserted into Java VM code using Dynamic Instrumentation. The tag types may also be required to generate different instrumentation code. For example, return values for SPARC routines are stored in a register, while return values for Java methods are pushed onto the method's operand stack. Instrumentation code that gets the return value from a Java method will differ from instrumentation code that gets the return value from a SPARC function. In this case, the type tag can be used to generate the correct code.

4.1.5 Java Interpreter-Specific Metrics

We have created several VM-specific metrics that measure aspects of the Java VM's execution of the Java application (Figure 4.4).

Java Specific Metric	What it measures
`loadTime`	Amount of wall clock time to load a Java .class file
`GCTime`	Amount of time the VM spends garbage collecting
`obj_create`	Amount of time due to VM handling object creates
`MethodCall_CS`	Amount of CPU time due to context switching on a method call

Figure 4.4 Java Interpreter Specific Metrics.

The `loadTime` metric is an example of a metric that needs to be measured for a particular Java application class before there are any resources created for the class; `loadTime` measures VM costs associated with Java application class file loading, and we do not parse and create resources for this class until after the class has been loaded by the VM. To obtain this measure, we instrument the VM routines that handle loading the Java application .class files, and transforming them from .class file format to expanded versions of the class that are stored in internal VM data structures. At the exit point of the VM loading routine, we collect the performance values and store them with the new resources that we create for this class. When a user makes a performance data query for `loadTime` associated with an AP class resource, we do not insert any instrumentation to collect the metric's value; instead, we just fetch the value that we already collected and stored with the AP class resource when it was loaded by the VM. When a user makes a performance query for the other metrics, Paradyn-J inserts instrumentation code into the AP and VM to collect the requested performance data.

4.1.6 Modifying the Performance Consultant to Search for Java Bottlenecks

One of Paradyn's main technologies is an automated performance bottleneck searcher, called the Performance Consultant. The Performance Consultant starts out searching for high level per-

formance bottlenecks by testing hypotheses based on performance data values. It refines hypotheses that test true to more specific explanations for the bottleneck and to more constrained parts of the executing program. The Performance Consultant searches for bottlenecks in three main categories: I/O, synchronization, and CPU. For traditional binary executions this is sufficient, however, for Java executions there is another important class of bottlenecks that should be added to the search: Java VM bottlenecks. One reason why a Java application's execution may be slow is because the AP triggers certain activities in the VM that are expensive. Two such activities, which we discovered from our performance measurement tests, are VM method call and VM object creation overhead. These activities occur whenever the VM interprets method call bytecode instructions and object creation instructions in the Java application. We modified the Performance consultant to search for VM method call overhead and VM object creation overhead if the high-level CPU bound hypothesis tests true for the application. With this modification, an AP developer can use the Performance Consultant to automatically search for VM-specific performance bottlenecks in the Java application program.

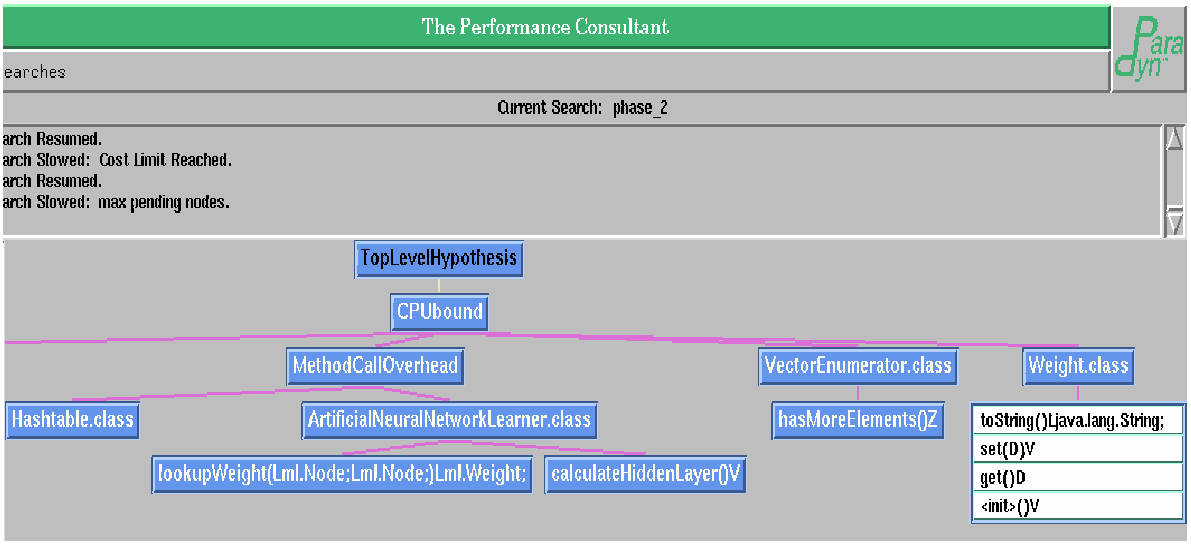

Figure 4.5 Performance Consultant search showing VM-specific bottlenecks in a neural network Java application. *Starting from the root node, the figure shows a bottleneck search that tested true for CPUbound (as indicated by a dark node), and a refinement of the search to test for the VM-specific bottleneck (MethodCallOverhead), and a further refinement of VM method call overhead to specific AP methods. The search also was able to account for portions of the CPU bottleneck in the Weight and VectorEnumerator classes. The white nodes represent untested hypotheses.*

An example of a Performance Consultant search for Java bottlenecks is shown in Figure 4.5. The figure shows a search history graph representing the Performance Consultants bottleneck search. From the root node the Performance Consultant first found that the interpreted execution was CPU bound (indicated by the darker shaded node labeled "CPUbound"). Next, the Performance Consultant tried to refine the search to particular VM-specific overhead that account for

part of the CPU time. In this case VM method call overhead account for a significant part of the CPU time. Finally, the Performance Consultant was able to account for a significant portion of the method call overhead due to two AP method functions `lookupWeight` and `calculateHiddenLayer` of the `ArtificialNeuralNetworkLearner` class. The search history graph also shows that the Performance Consultant was also able to isolate significant portions of the CPU time to individual AP classes and methods (the VectorEnumerator and Weight classes).

4.2 Transformational Instrumentation Costs

We analyzed instrumentation overhead associated with Paradyn-J's Transformational Instrumentation. Results of our measurements show that the generation and insertion of Java byte-code instrumentation at run time is comparable to Paradyn's dynamic instrumentation. However, application perturbation due to executing Java byte-code instrumentation can be high.

Figure 4.6 shows timing measurements of Paradyn-J's handling of an instrumentation request from a user. The measures were taken from Paradyn-J's implementation for version 1.1.6 of JDK running on an UltraSparc 30 running Solaris 2.6. The figure shows the costs of instrumenting the metric `procedureCalls` for a medium-sized method (`Device.start`). Paradyn-J will relocate `Device.start` to the heap with holes, generating and inserting SPARC instrumentation, and inserting a method call byte-code instruction at the entry point of the relocated method that jumps to `do_baseTramp`. If this is the first instrumentation request for a method in the `Device` class, then the class' constantpool is also modified with entries which resolve the method byte-code instruction that calls `do_baseTramp`. The `Device.start` method has 100 byte-code instructions, is 142 total bytes, and has 17 method call instructions requiring 36 holes when it is relocated to the heap (one at the method's entry and exit, and one before and after each call instruction). The time to parse the metric-focus pair and to create an intermediate form (called an AST node) that will be used to generate SPARC instrumentation code, is a cost that is present in Paradyn's Dynamic Instrumentation[1]. The other costs are unique to Transformational Instrumentation. The timing measures show that Transformational instrumentation adds a relatively small amount of overhead to the costs already present in Paradyn's Dynamic Instrumentation.

Figure 4.7 shows timing measurements of Transformational Instrumentation's perturbation: the amount of time spent executing instrumentation code in an instrumented AP byte-code method. The timing measures show the time Paradyn-J spends in instrumentation code for: (1) the VM to interpret the method call instruction that jumps to the SPARC instrumentation[2], and (2) SPARC instrumentation (including the time spent in the native method `do_baseTramp`). The costs were measured for both counter and timer instrumentation. Executing timer instrumenta-

1. 70% of the 67.5 ms, is due to a routine that creates lists of meta-data used to create the AST node, the other 30% is due to parsing the metric-focus pair, creating the AST node, and allocating counters and timers in a shared memory segment

Measurement		Time
	Time to relocate Device.start to the heap with holes	9.4 ms
	Time to generate and insert instrumentation at one of method's instrumentation point (including generating and inserting byte-code call to do_baseTramp, and code in do_baseTramp to jump to base tramp and mini tramp instrumentation)	1.6 ms
	Time to add do_baseTramp entries to the Device class' constantpool	1.5 ms
	Time to parse metric/focus pair and create an intermediate form (an AST node)	67.5 ms
Total time to handle an instrumentation request		**84.1 ms**

Figure 4.6 Timing measures for a Transformational Instrumentation request. *The instrumentation request is for the metric procedureCalls for a medium sized method function Device.start. The time to parse the metric/focus pair and create the AST node is a cost that is present in Paradyn's dynamic instrumentation. The other costs are unique to Transformational Instrumentation.*

Measurement		Time
	Time for VM to interpret a call to and return from an empty static native method with one integer parameter (the cost just to call to and return from do_baseTramp(int i))	3.6 µs
	Time spent in our instrumentation routine (do_baseTramp(int i)) including SPARC instrumentation code with one timer read	7.6 µs
Total Time spent executing instrumentation with one timer mini-tramp		**11.2 µs**
	Time for VM to interpret just the call to and return from do_baseTramp(int i)	3.6 µs
	Time spent in our instrumentation routine (do_baseTramp(int i)) including SPARC instrumentation code with one counter read	2.6 µs
Total Time spent executing instrumentation with one counter mini-tramp		**6.2 µs**

Figure 4.7 Timing measures of Transformational Instrumentation perturbation. *The number of seconds spent in our instrumentation code at each instrumentation point, and a break down of this time into the time for the VM to interpret the method call instruction that jumps to the SPARC instrumentation, and the amount of time spent in the SPARC instrumentation (the time spent in the native method do_baseTramp including base tramp and mini tramp code).*

tion code is more expensive than executing counter code (7.6 µs vs. 2.6 µs), because timer instrumentation makes a system call to obtain process time. The results show that the VM's interpretation of the method call instruction to `do_baseTramp` accounts for 58% of the cost of executing counter instrumentation code (3.6 µs of 6.2 total µs), and accounts for 32% of the instrumentation's execution time for timer instrumentation (3.6 µs of 11.2 total µs). The interpreted call instruction from an instrumentation point to the SPARC instrumentation code is an expensive part of Transformational Instrumentation.

2. The measures include only the execution of Sun's quick version of the method call instruction; the first time the call instruction is interpreted the VM replaces regular call instruction with their quick version and modifies the class' constantpool entries to point to the target method (this may trigger class file loading). We do not include these extra costs in our timings.

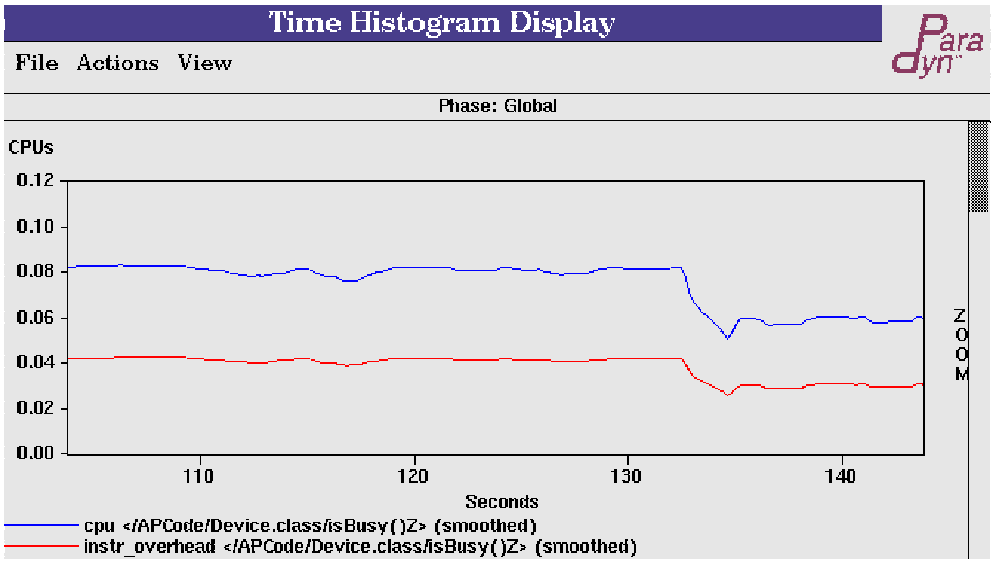

Figure 4.8 Performance Data showing part of transformational instrumentation perturbation. *The time histogram show the amount of time spent in most of our instrumentation code (excluding just the time for the VM to interpret the method call instruction to jump to our instrumentation) to measure CPU time for method Device.isBusy. This shows that approximately 1/4 of the method's execution time is spent executing instrumentation code (1/2 of the 0.04 seconds per second of execution time is included in the measure of CPU time for Device. isBusy). Device.isBusy has two instrumentation points for its 3 original byte-code instructions.*

Finally, in Figure 4.8 we show performance data collected by Paradyn-J that measure part of Paradyn-J's instrumentation perturbation costs. In this example we show the amount of time spent in our instrumentation routine `do_BaseTramp` versus the amount of CPU time the instrumentation measures for a method (the time spent in `do_baseTramp` accounts for most of the overhead of Transformational Instrumentation). These measures provide only a rough idea of how much our instrumentation perturbs the performance data computed by the instrumentation because we are using instrumentation to measure the perturbation. As a result, about one half of the time spent in the instrumentation code will be included in the performance measure. We picked a small simple method to exacerbate the instrumentation perturbation. In general, the amount of perturbation varies depending on the frequency of execution of the method's instructions that are instrumented; for methods with a few simple byte-code instructions, instrumentation code execution will dominate the method's execution time and, as a result, perturb the performance measures more. One place where this can cause a problem is in the Performance Consultant's bottleneck search. Here a large amount of perturbation can result in method functions incorrectly testing true for a particular bottleneck.

4.3 Advantages and Disadvantages of Transformational Instrumentation

The advantages of Transformational Instrumentation are that unmodified Java AP byte-

codes can be instrumented, it does not require a special version of the Java VM to obtain measurements of AP byte-codes, and instrumentation code can be inserted or deleted from AP byte-codes at any point in the interpreted execution. The disadvantages of Transformational Instrumentation are that it requires a non-trivial effort to port to new virtual machines, and that it is expensive to execute because of the interpreted method call used to jump to instrumentation code. Only with changes to the Java instruction set and JDK interfaces for obtaining AP CPU times, can fast byte-code instrumentation be implemented.

One nice feature of Transformational Instrumentation is that it allows Paradyn-J to wait until run-time to instrument Java byte-codes, thus requiring no changes to Java .class files or source code. As a result, Paradyn-J can measure Java Class Library code just like any other Java application code. Also, because we wait until run-time to instrument the Java VM, we did not have to build a special version of the VM to obtain AP performance measures. Because instrumentation code can be added or removed from byte-codes at any point in the execution, run-time perturbation can be controlled by only inserting instrumentation to collect the performance data that the tool user requested.

One disadvantage of our instrumentation approach is that porting Paradyn-J to a new VM is complicated by instrumenting byte-codes and modifying class constantpools after they have been loaded into VM data structures. Since these data structures differ between VM implementations, Transformational Instrumentation is non-trivial to port to a new virtual machine (in Chapter 7 we discuss porting issues associated with Paradyn-J in more detail).

The biggest disadvantage of our instrumentation technique is that it is expensive to execute, and can result in large perturbations of a method's execution. It is expensive to execute because of the interpreted method call that jumps to the SPARC instrumentation code.

Ideally, we would like to use all byte-code instrumentation to instrument Java AP byte-codes. However, the only way to inexpensively instrument Java byte-codes with all byte-code instrumentation is if changes to the Java instruction set are made to add instructions to explicitly create and restore execution stack frames and method operand stacks, and if JDK provided fast access for obtaining per AP thread CPU times. With new Java instructions to create a new operand stack and create and restore an execution stack frame, we would no longer need to add `nop` holes around instrumentation points in an AP method; we could safely overwrite method byte-code instructions with jumps to our instrumentation code and relocate the overwritten byte-code instructions to the instrumentation code in the heap. Byte-code instructions to create a new operand stack and execution stack frame could be added at the beginning of our instrumentation code, thus eliminating the need to use an expensive method call instruction to jump to byte-code instrumentation. Also, by inserting instructions to restore the method's execution stack frame before the relocated method instructions are executed, the method's relocated instructions can correctly execute using the method's operand stack and within its own stack frame. As a result, we would

no longer need relocate a method to the heap with `nop` holes around its instrumentation points. Thus, performance of executing byte-code instrumentation would be further improved by getting rid of the `goto_w` instruction that jumps from the old method to the version of the method with `nop` holes.

Not until the Java 2 Platform release has there been a way to obtain thread-level CPU times (via the JVMPI interface [62]). However, the CPU times are provided by JVMPI only on method entry and exit. As a result, the amount of time spent in a method by a particular thread can be correctly computed only when the thread exits the method, which does not fit with our model of sampling a thread's timer to obtain performance data during the thread's execution of the method. Also, as we discuss in Chapter 7, obtaining CPU measures through JVMPI can be costly and JVMPI does not provide enough functionality to completely implement our model. With the current state of JDK, we can trade-off the portability of the JVMPI interface against the generality of our measurement techniques.

4.4 Performance Tuning Study of an Interpreted Java Application

We used performance data from Paradyn-J to determine how to tune the application to make it run faster. In this study, we are using Paradyn-J for JDK version 1.0.2 running on a UltraSparc-Solaris2.6 platform. The application consists of eleven Java classes and approximately 1200 lines of Java code. Figure 4.9 shows the resource hierarchies from the interpreted execution, including the separate AP and VM code hierarchies (**APCode** and **Code**).

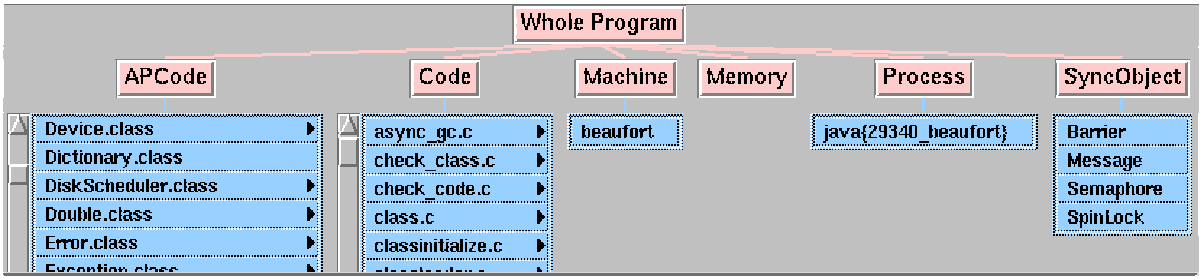

Figure 4.9 Resource hierarchies from interpreted Java execution.

We took the view of the Java application developer to show how to use Paradyn-J to improve the performance of an interpreted application. We first tried to get an idea of the high-level performance characteristics. A program's execution can be characterized by four basic costs, I/O time, synchronization time, memory access time[1], and CPU time. Since the application is single

1. Currently, the production Solaris version of Paradyn does not include metrics for memory profiling, and we did not add memory profiling metrics for Paradyn-J's implementation. However, for systems that provide user level access for obtaining memory hierarchy activity, new memory profiling metrics can be easily added to Paradyn and Paradyn-J via Paradyn's Metric Description Language [28].

threaded, we know there is no synchronization waiting time in its execution. Figure 4.10 is a time plot visualization showing the fraction of time the application spends on I/O and CPU. The top curve is the fraction of CPU time per second execution time (about 98%) plotted over the interpreted execution. The bottom curve is the fraction of I/O blocking time per second execution time (about 2% of the total execution time)[1].

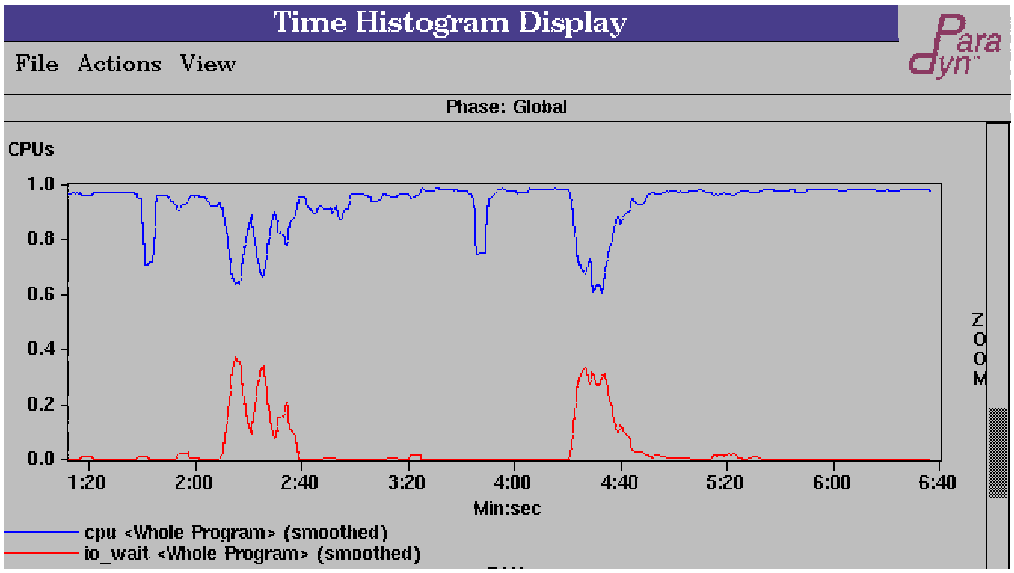

Figure 4.10 High-level performance characteristics of the interpreted Java program. *The interpreted execution can be dominated by I/O, synchronization or CPU times. In this case the execution is CPU bound.*

We next tried to account for the part of the CPU time due to specific VM costs associated with the Java interpreter's execution of the Java application. In some of our initial tests of Paradyn-J, we measured different VM functions; as a result, we discovered that the VM's handling of method calls and object creates are expensive activities, so we want to account for part of the CPU time in terms of these two VM activities. To measure these VM costs, we used two VM-specific metrics: **MethodCall_CS** measures the time for the Java VM to handle a method call, and **obj_create** measures the time for the Java VM to create a new object. Both are a result of the VM interpreting specific byte-code instructions in the AP (method call and object creation instructions). We measured these values for the Whole Program focus (no constraints), and we found that a large portion (35%) of the total CPU time is spent handling method calls and object creates (Figure 4.11).

Figure 4.11 suggests that we might improve the interpreted execution by reducing the VM method call and object creation overhead. Since we are tuning the AP and not the VM in this

1. Exact numerical values were obtained from a tabular display of the same performance data.

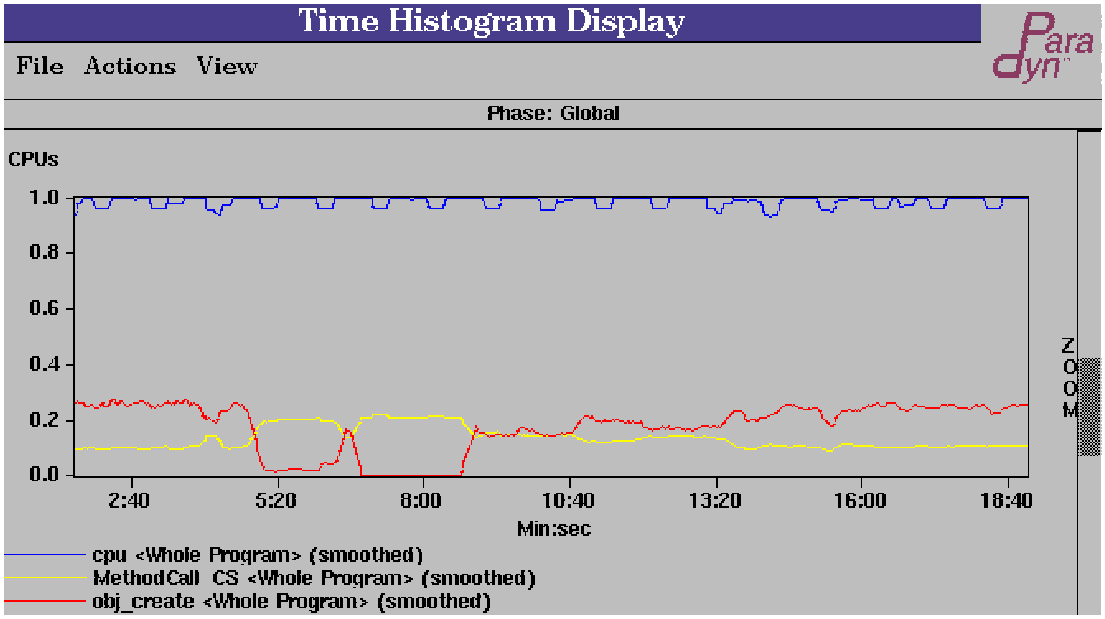

Figure 4.11 Performance data showing VM overhead associated with the Java application's execution. *Method call overhead (yellow curve), object creation overhead (red curve), and cpu time (blue curve).*

study, we modified the AP program to remove some method calls and object creates from its execution.

We first tried to reduce the method call time by in-lining some method functions in the AP. To allow our tuning efforts to have the largest effect on the program's execution, we should focus our tuning efforts on those parts of the AP in which we are spending the most time. Figure 4.12 is a table visualization showing the AP methods that are accounting for the largest fraction of CPU time. The first column of the table lists constrained parts of the Java application (individual methods and classes), and the second column lists a metric (fraction of CPU time) associated with each method or class. The `Sim` and `Device` classes account for the largest fraction of CPU time: 30% and 12% respectively.

Figure 4.13 shows CPU and VM method call times. The time plot visualization shows the fraction of CPU time spent in the `Sim` class (30%), the fraction of CPU time due to VM method call overhead (20%), and the interaction between the two: the fraction of VM method call overhead due to the VM interpreting method call instructions in `Sim` class byte-codes (5% of the total execution or about 20% of the `Sim` class's execution).

At this point, we know how to tune the AP program (remove some method calls), and where to perform the tuning (methods in the `Sim` and `Device class`). However, we still do not know which method calls to in-line in `Sim` and `Device` method functions. Figure 4.14 helps the us answer the question of which methods to in-line; it shows the method functions that are called

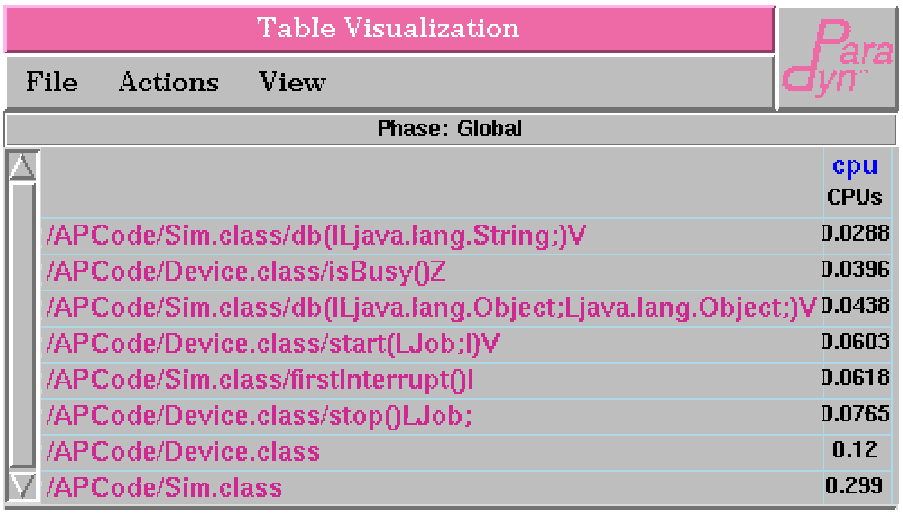

Figure 4.12 The fraction of CPU time spent in different AP methods. *The first column of the table lists constrained parts of the Java application, and the second column is the fraction of CPU time spent in each method. The table is sorted bottom-up (30% of the time we are executing methods in the Sim class).*

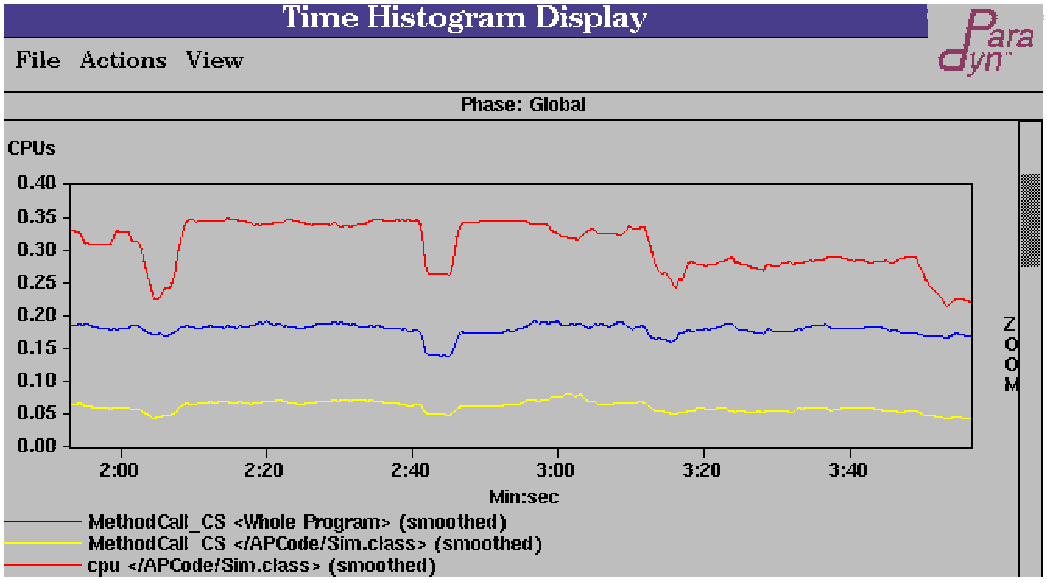

Figure 4.13 VM method call overhead associated with the Sim.class. *The time plot shows the fraction of CPU time spent interpreting Sim class byte-codes (top curve), the fraction of CPU time the VM spends handling method calls (middle curve), and the fraction of CPU time the VM spends handling method calls due to interpreting method call byte-code instructions in the Sim class (bottom curve).*

most frequently. If we in-line calls to these methods, then we will remove a large number of method calls in the program execution and, as a result, we will reduce the largest amount of VM method call overhead.

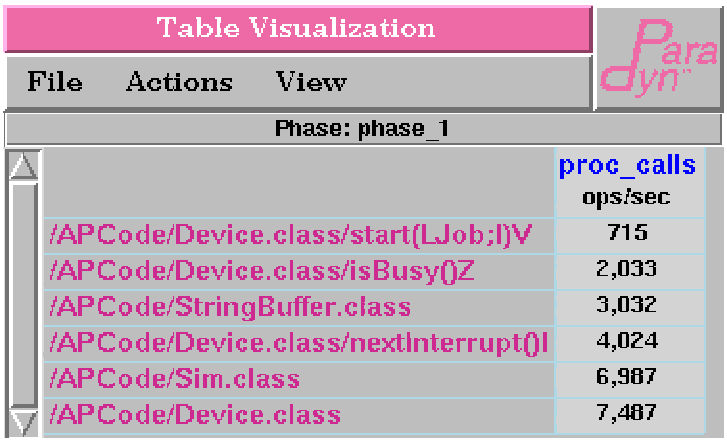

Figure 4.14 Performance Data showing which methods are called most frequently. *Device.isBusy and Device.nextInterrupt are called most frequently and account for almost all of the 7,487 calls/second to methods in the Device class.*

Figure 4.14 shows us that the `nextInterrupt` and `isBusy` methods of the `Device` class are being called most frequently. By examining the code we found that the `Sim.now` method was also called frequently (this is a method that we cannot currently instrument with Paradyn-J because it is too small). These three methods are all small, they all return the value of a private data member, and they are all called frequently, thus they are good candidates for in-lining. After changing the AP code to in-line calls to these three methods, we were able to reduce the total number of method calls by 36% and the total execution time by 12% (second row in Table 4.15).

Optimization	Number of Method Calls	Number of Object Creates	Total Execution Time (in seconds)
Original Version	`44,561,130`	`1,484,430`	`286.85`
Method in-lining	`28,382,051 (-36%)`	`1,484,430`	`253.75 (-12%)`
Fewer Obj. Creates	`35,531,280 (-20%)`	`53,682(-96%)`	`201.71 (-30%)`
Both Changes	`19,352,201 (-57%)`	`53,682`	`170.61 (-41%)`

Figure 4.15 Performance results from different versions of the application.

We next tried to reduce the number of VM object creates by finding the AP methods where most of the objects are created. Figure 4.16 shows performance data associated with parts of the Java application program (individual methods and classes). The table shows the AP methods that are creating the most objects. The metric **num_obj_create** measures the number of objects created per second in each method and class. We can see from this data that most objects are created in the `Job`, `Device` and `Sim` class methods. Of these, the `Job.toString`, the `Device.stop` and the `Device.start` methods create the most objects.

Table Visualization

File Actions View

Phase: Global

	num_obj_create ops/sec
/APCode/Job.class/<init>(Ljava.io.DataInputStream;)V	15.17
/APCode/Sim.class/db(ILjava.lang.Object;Ljava.lang.Object;Ljava.lang.Object;Ljava.lang.Object;)V	17.51
/APCode/Sim.class/schedule(LJob;LDevice;)V	72.5
/APCode/Device.class/stop()LJob;	261.9
/APCode/Device.class/start(LJob;I)V	261.9
/APCode/Device.class	523.8
/APCode/Job.class/toString()Ljava.lang.String;	739.7
/APCode/Job.class	756.1

Figure 4.16 Table showing the number of objects created/second in AP classes and methods. *The table is sorted bottom-up. Most objects are created in the Job, Device and Sim class methods.*

The data in Figure 4.16 tell us from which AP methods object creates should be removed. However, we still do not know which object creates we should try to remove; we need to know which kind of objects are created most frequently. Figure 4.17 shows the types of objects that are being created most frequently in the execution. The table shows the number of calls per second to constructors of the `String`, `StringBuffer`, and `Job` classes. The performance data in Figure 4.16 and Figure 4.17 tell us to look for places in the `Job.toString`, the `Device.stop` and the `Device.start` where `String` and `StringBuffer` object creates can be removed.

On examining code in the `Device.start` and `Device.stop` methods, we were able to reduce the number of `StringBuffer` and `String` objects created by removing strings that were created but never used, and by creating static data members for parts of strings that were recreated multiple times. By modifying the original Java application program in this way, we are able to reduce the total execution time by 30% (third row in Figure 4.15). A version of the application modified by removing both method calls and object creates results in a reduction of total execution time by 41%.

In this example, Paradyn-J provided performance data that is difficult to obtain with other performance tools (performance data that describe expensive interactions between the Java VM and the Java application). Paradyn-J presents performance data describing specific VM-AP interactions, such as VM method call overhead associated with the VM interpreting call instructions in `Sim` class methods. With this data, we were easily able to determine what changes to make to the Java application and improve its performance by a factor of 1.5.

We took the view of the AP developer using performance data from Paradyn-J; Paradyn-J's performance data helped us to tune the Java AP by modifying the AP to reduce the number of method calls and object creates in its execution. Performance data from Paradyn-J also is useful

Table Visualization

File Actions View

Phase: Global

	procedure_calls ops/sec
/APCode/Job.class/<init>(Ljava.io.DataInputStream;)V	2.316
/APCode/String.class/<init>([B)V	4.679
/APCode/String.class/<init>([BII)V	4.679
/APCode/String.class/<init>(II[C)V	9.674
/APCode/StringBuffer.class/<init>(I)V	661.3
/APCode/String.class/<init>(Ljava.lang.StringBuffer;)V	661.3
/APCode/StringBuffer.class/<init>()V	661.3
/APCode/String.class/<init>([CII)V	661.7

Figure 4.17 Performance data showing which objects are created most frequently. *Table visualization showing number of calls/second to constructors of the String, StringBuffer, and Job classes. This shows that we are creating a lot of String and StringBuffer objects.*

to the Java VM developer. Performance data that show that interpreting object creation code and interpreting method call instructions in the AP is expensive, tell the VM developer to focus on tuning the VM routines responsible for handling method calls and object creates to try to reduce these costs. Also, performance data from Paradyn-J that associates the time spent in VM method call and object creation routines with specific AP code, will help an VM developer focus in on certain types of method calls or object creates that are more expensive than others and, as a result, possibly modify the VM routines to handle these special cases better.

4.5 Conclusions

In this chapter, we presented Paradyn-J, a prototype performance tool for measuring interpreted Java executions. Paradyn-J is an implementation of our model for describing performance data from interpreted executions. In a performance tuning study of a Java application program, we demonstrated that the type of performance data that can easily be described by performance tools based on our model, allows a program developer to answer questions about how to tune the AP to improve its performance when run on a specific VM. For example, we showed how performance data describing specific VM-AP interactions such as VM method call overhead in the `Sim` class of the Java application, lead us to determine how to tune the program (by in-lining some method calls), and where to focus our tuning efforts (in-lining method calls in the `Sim` and `Device` classes of the Java application).

Chapter 5

Motivational Example

In an interpreted execution, there are many interactions between the VM and the AP; it is apparent that performance data from interpreted executions that represent VM costs associated with an AP's execution is useful to a program developer. We know that profiling dynamically compiled native code is useful (as is profiling any native code); however, for dynamically compiled executions it is not as obvious that performance data that describe specific interactions between a VM and AP code is necessary.

We present results from a performance study comparing a dynamically compiled execution to an all-interpreted execution of three Java application kernels. The results of this study motivate the need for a performance tool for dynamically compiled Java executions. We demonstrate that performance data that describe specific VM costs associated with a method that is compiled at run-time is critical to understanding the method's performance.

5.1 Performance Measurement Study

We compare total execution times of dynamically compiled and all-interpreted executions of three Java applications. We examine three cases where we suspect that the performance of dynamic compilation and subsequent direct execution of a native form of a method might be the same as, or worse than, simply interpreting a byte-code version of the method: (1) methods whose native code interacts frequently with the VM, (2) methods whose execution time is not dominated by executing method code (e.g., I/O intensive methods), and (3) small methods with simple byte-code instructions.

The performance of a dynamically compiled Java method can be represented as the sum of the time to interpret the byte-code form, the time to compile the byte-code to native code, and the time to execute the native form of the method:
$a \times Interp + Compile + b \times NativeEx$ (where $a + b = n$ is the number of times the method is executed). We examine three cases where we suspect that the cost of interpreting a method is

less than the cost of dynamically compiling it: where
$(n \times Interp) < (a \times Interp + Compile + b \times NativeEx)$. We implemented three Java application kernels to test these cases. Each kernel consists of a main loop method that makes calls to methods implementing one of the three cases. We ran each application for varying numbers of iterations under Sun's ExactVM, which is part of the Java 2 Platform release of JDK [63], and compared executions with dynamic compiling disabled to executions that used dynamic compiling. ExactVM uses a count based heuristic to determine when to compile a method; if the method contains a loop it is compiled immediately, otherwise ExactVM waits to compile a method until it has been called 15 times. As a result, the main loop method is immediately compiled, and the methods called by the main loop are interpreted the first 14 times they are called. On the 15th call, the methods are compiled and directly executed as native code for this and all subsequent calls. Calls from the native code in the main loop to the byte-code versions of the methods require interaction with the VM. Calls from the native code in the main loop to native code versions of the methods involve no VM interactions.

Case 1: Methods with VM interactions: The execution of the native form of the method can be dominated by interactions with the VM. Some examples include methods that perform object creation, deletion (resulting in increased garbage collection), or modification (either modifying an object pointer, or modifications that have side effects like memory allocation), and methods that contain calls to methods in byte-code form. To test this case, we implemented a Java application kernel that consists of a main loop that calls two methods. The first method creates two objects and adds them to a Vector, and the second method removes an object from the Vector. After each main loop iteration, the Vector's size increases by one. The Java Class Libraries' Vector class stores an array of objects in a contiguous chunk of memory. In our application, there are VM interactions associated with the two objects created in the first method. The increasing size of the vector will result in periodic interactions with the VM: when an object is added to a full Vector, the VM will be involved in allocating a new chunk of memory, and in copying the old Vector's contents to this new chunk. Object removal will result in increased garbage collection activity in the VM, as the amount of freed space increases with each main loop iteration. Our hypothesis is that the dynamic compilation of methods that create, modify, and delete objects will not result in much improvement over an all-interpreted execution because their execution times are dominated by interactions with the VM.

Results are shown as Case 1 in Table 5.1. For about the first 3,000 iterations, interpreted execution performs better than dynamically compiled execution. After this, the costs of runtime compilation are recovered, and dynamic compilation performs better. However, there are no great improvements in the dynamically compiled performance as the number of iterations increase. This is due to VM interactions with the native code due to object creates and modifications[1]. Also, the decrease in speed-up values between 10,000 and 100,000 iterations is due to an increase in the amount of VM interaction caused by larger Vector copies and more garbage collection in the

100,000 iteration case. Each method's native execution consists of part direct execution of native code and part VM interaction; in the formula from Section 5.1, the $b \times NativeEx$ term can be written as $b \times (DirectEx + VMInteraction)$. In this application, the *VMInteraction* term dominates this expression, and as a result, dynamic compilation does not result in much performance improvement.

Case 1: object modifications				Case 2: I/O intensive				Case 3: small methods			
iters	**Dyn**	**Intrp**	**S-up**	**iters**	**Dyn**	**Intrp**	**S-up**	**iterations**	**Dyn**	**Intrp**	**S- up**
100,000	114.7	119.5	1.04	100,000	427.1	436.43	1.02	10,000,000	1.76	35.11	19.94
10,000	1.73	2.04	1.18	10,000	40.47	42.70	1.05	1,000,000	0.83	4.16	5.01
1,000	0.71	0.65	0.91	1,000	4.53	4.64	1.02	100,000	0.74	0.98	1.32
100	0.70	0.63	0.90	100	1.06	0.99	0.94	10,000	0.72	0.67	0.93
								1,000	0.73	0.63	0.86

Figure 5.1 Execution time (in seconds) of each Java kernel run by ExactVM comparing interpreted Java (*Intrp* column) to dynamically compiled Java (*Dyn* column). *The number of iterations of the kernel's main loop is listed in the **iters** column, and speed ups are listed in the **S-up** column. Each measurement is the average of 10 runs.*

Performance data that represent VM costs of object creation and modification, and can associate these costs with particular AP methods, can be used by an AP developer to tune the AP. For example, if performance data verifies that VM object creation costs dominate the execution of the native and byte-code forms of a method, then the AP developer could try to move to a more static structure.

Case 2: Methods whose performance is not dominated by interpreting byte-code: A method's execution time can be dominated by costs other than executing code (e.g., I/O or synchronization costs). For this case, we implemented a Java application kernel consisting of a main loop method that calls a method to read a line from an input file, and then calls a method to write the line to an output file. We use the Java Class Library's `DataOutputStream` and `DataInputStream` classes to read and write to files. Our hypothesis is that dynamic compilation of the read and write methods will not result in much improvement because their native code execution is dominated by I/O costs.

The results of comparing an interpreted to a dynamically compiled execution on different sized input files (the number of lines in the input file determines the number of main loop iterations) are shown as Case 2 in Table 5.1. After about 500 iterations, the dynamically compiled execution performs better than the all-interpreted execution. Speed-ups obtained for an increasing

1. We verified this by measuring an all-interpreted and a dynamically compiled execution for a similarly structured application kernel without object creates, modifies or deletes. Speed ups show dynamic compilation results in better performance as the number of iterations increase (for 100,000 iterations a speedup of 4.9 vs. a speed up of 1.04 for Case 1)

number of iterations are not that great; I/O costs dominate the native code's execution time[1]. The decrease in speed-up values between the 10,000 and 100,000 iteration case is due two factors. First, each read or write system call takes longer on average (about 3.5%) in the 100,000 case, because indirect blocks are used when accessing the input and output files. Second, there is an increase in the amount of VM interaction caused by garbage collection of temporary objects created in `DataOutputStream` and `DataInputStream` methods; for larger files, more temporary objects are created and, as a result, VM garbage collection activity increases.

Performance data that represent I/O costs associated with a method's execution could be used by an AP developer to tune the AP. For example, performance data that indicate a method's execution time is dominated by performing several small writes could be used by an AP developer to reduce the number of writes (possibly by buffering) and, as a result, reduce these I/O costs.

Case 3: Methods with a few simple byte-code instructions: For small methods, the time spent interpreting method byte-codes is small, so the execution of the native form of the method may not result in much improvement. To test this case, we wrote a Java application kernel with a main loop method that calls three small methods; two change the value of a data member and one returns the value of a data member. Our hypothesis is that dynamic compilation of these three small methods will not result in much improvement because their interpreted execution is not that expensive.

The results (Case 3 in Figure 5.1) show that there are a non-trivial number of iterations (about 25,000) where an all-interpreted execution outperforms a dynamically compiled execution. However, as the number of iterations increases, the penalty for continuing to interpret is high, partly because of the high overhead for the VM to interpret method call instructions vs. the cost of directly executing a native code call instruction[2]. Performance data that explicitly represent VM method call overheads, VM costs to interpret byte-codes, and VM costs to execute native code could be used by an AP developer to identify that interpreted call instructions are expensive.

5.2 Discussion

The result of this study points to specific examples where detailed performance measures from dynamically compiled executions provides information that is critical to understanding the execution. For real Java applications consisting of thousands of methods, some with complicated

1. We verified this by measuring an all-interpreted and a dynamically compiled execution for a similarly structured application kernel without I/O activity. Speed ups show dynamic compilation results in better performance as the number of iterations increase (for 100,000 iterations a speedup of 4.9 vs. a speed up of 1.02 for Case 2)
2. We verified this by measuring an all-interpreted and a dynamically compiled execution of a similarly structured application kernel that makes calls to empty methods (the cost of executing the method is just the VM overheads to handle method calls and returns). For 10,000,0000 iterations there was a speed up of 31.8, and for a version with no method call overheads (all code is in the main loop) a speed up of 11.2.

control flow structure, a performance tool that can represent specific VM costs (like method call, and object creation overheads) and I/O costs associated with byte-code and native code can be used by an AP developer to more easily determine which AP methods to tune and how to tune them.

A VM developer can also use this type of performance data to tune the VM. For example, performance data that describes VM object creation overhead associated the byte-code form and native code form of an AP method, tell the VM developer to focus on tuning the VM routines that handle object creates. Another way in which a VM developer could use performance data is to incorporate it into the dynamic compiler's runtime compiling heuristic. Performance data collected at run-time can be used to trigger a method's run-time compilation, to exclude a method from run-time compilation, or to produce better optimized code for a run-time compiled method. In Chapter 6 we discuss further implications of this study for AP and VM developers.

Chapter 6

Describing Performance Data from Applications with Multiple Execution Forms

We present a representational model for describing performance data from an application program with multiple execution forms. Our model addresses two problems: (1) how to represent the multiple execution forms of AP code objects that are transformed at run-time, and (2) how to map performance data between different execution forms of an AP code object. Also, we present a proof-of-concept implementation of our model: modifications to Paradyn-J to add support for measuring dynamically compiled Java programs. In a performance tuning study of a dynamically compiled Java application, we show how performance data from Paradyn-J can be used to tune an AP method and improve its performance by 10%.

6.1 Representing the Application's Multiple Execution Forms

Our model describes performance data associated with AP code objects that change form during execution. Our goals are to represent the different forms of a transformed object, represent the relationship between the different forms of an AP object, map performance data between different views (between different forms) of a transformed object, and represent performance measures associated with the run-time transformations of AP code objects.

6.1.1 Representing Different Forms of an AP Code Object

AP code can be executed in more than one form. For example, methods from dynamically compiled Java applications can have two execution forms: an interpreted byte-code form and a native code form. We represent each execution form of an AP code object as a resource instance in the `APCode` hierarchy. For example, a Java method `foo` has an initial byte-code form resource instance. When it is compiled at run-time, a new native code form resource is created for its compiled form and added to the `APCode` hierarchy. In other words, run-time transformations of AP code objects create new resource instances in the `APCode` hierarchy.

We represent the different types of AP code objects as instances of **Class**, **Method**, **Module**, and **Function** class resources; the `APCode` resource hierarchy can have children of either the **Class** (byte-code) type or the **Module** (native code) type or both. An example from a Java AP is shown in Figure 6.1. The figure shows resource instances corresponding to a Java class `Blah`, which contains byte-code method functions `foo` and `blah`, and a native method `g`, which is contained in shared object `libblah.so`. The resources are discovered when the Java VM loads AP class or shared object files at run-time.

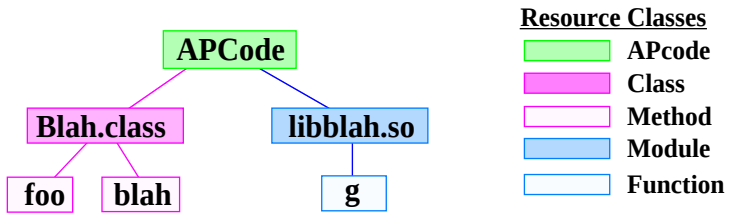

Figure 6.1 Types of resource instances that the APCode hierarchy can contain. For a Java application the APCode hierarchy can contain resource instances of byte-code and native code classes.

When a byte-code method is compiled at run-time, a new **Function** resource instance is added to the `APCode` hierarchy for the compiled form. For example, in Figure 6.2, method `foo` is compiled at run-time to native code. A new **Function** resource instance `foo` is created for the compiled form, and a new **Module** resource instance `Blah_native` is created as its parent (the parent of all of the compiled form methods of the `Blah.class`).

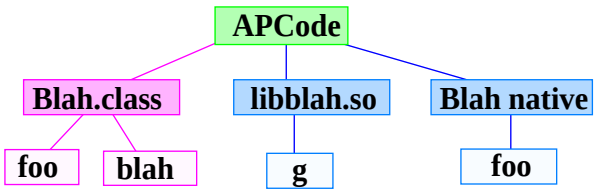

Figure 6.2 The APCode hierarchy after method `foo` is compiled at run-time. If `foo` is compiled at run-time into native code, we create a new **Function** resource for its native code form and a new **Module** resource corresponding to the `Blah` class' run-time compiled forms (`Blah_native`).

There is a relationship between some AP byte-code and native form resource instances; some byte-code resources are compiled into native code. *Resource mapping functions* are discovered when AP methods are compiled at run-time and are used to map between byte-code and native-code forms of resources. We use resource mapping functions to map performance data between the multiple execution forms of AP code objects, and to generate instrumentation for dynamically compiled code.

6.1.2 Resource Mapping Functions

We define resource mapping functions and show how they are used to map performance data requests made in terms of AP byte-code constraints to instrumentation of the corresponding AP native code constraints, and to map performance data collected for AP native code back to the original AP byte-code. Also, we discuss mapping performance data in the presence of complications caused by optimizing compilers; many-to-one resource mappings can result from method in-lining either because resource mapping granularities are too coarse, or because parts of the resulting native code cannot be attributed to individual AP byte-code methods from which the native code was compiled.

- Definition 6.1: A *resource mapping function, f,* is a partial function that maps a sub-set of resource instances (constraints) in one execution form to sub-set of corresponding resource instances in another execution form: f: $2^S \rightarrow 2^S$, where S is the set of all constraints in `APCode`, and 2^S is the set of all sub-sets of S.

Mapping functions are built-up incrementally during the program execution; as an AP code object is transformed at run-time, new mappings are discovered. In Figure 6.2, method function `foo` is run-time compiled to native code. The resulting resource mapping function are:

f: {`/APCode/Blah.class/foo`} → {`/APCode/Blah_native/foo`}, and

f: {`/APCode/Blah_native/foo`} → {`/APCode/Blah.class/foo`}.

Mapping performance data and instrumentation between execution forms of AP code is easy for one-to-one and one-to-many transformations. However, it is more complicated for many-to-one and many-to-many transformations. One-to-one resource mappings result from the compilation of byte-code method `foo` to native code function `foo_n` (Figure 6.3). The resource mapping f:{`/APCode/Blah.class/foo`} → {`/APCode/Blah_native/foo_n`} is used to map data requests for constraint `foo` to instrumentation of `foo_n`, and the resource mapping f:{`/APCode/Blah.class/foo_n`} → {`/APCode/Blah_native/foo`} is used to map performance data collected for the native form of the method (`foo_n`) back to be viewed in its byte-code form (`foo`).

One-to-many transformations can occur for various types of compiler optimizations including specialization, and method in-lining. In specialization, the Java VM collects run-time information (e.g., method parameter values), and uses these to create specialized version(s) of the compiled method. The example in Figure 6.4 shows a transformation of one byte-code method, `blah`, into two specialized versions of the method in native code form (`blah_10`, and `blah_20`). A 1-to-2 resource mapping results from the transformation of `blah` to {`blah_10`, `blah_20`}, and two 1-to-1 mappings map each native form to `blah`. Performance data requests for `blah` are mapped to instrumentation of its two native code forms (`blah_10` and `blah_20`), and performance data collected for the two native code forms are mapped back and aggregated (e.g. summed or averaged) into a single value that is viewed in terms of `foo`.

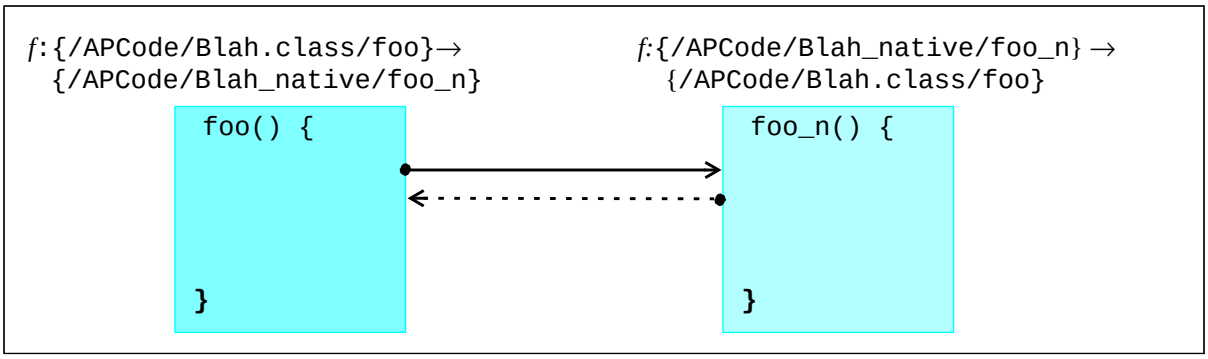

Figure 6.3 Example of a 1-to-1 resource mapping : *A 1-to-1 mapping occurs when method* `foo` *is compiled at runtime to a native code version (*`foo_n`*). The solid arrows indicate a mapping form the byte-code resource instance to its native code form, and the dotted arrow indicates a mapping from a native code resource instance to its byte-code form.*

Method in-lining also can result in 1-to-N mappings. In-lining is a transformation of one method `foo` that calls method `blah` into a native code version of the method (function `foo_n`) with the call instruction replaced with in-lined `blah` code (bottom of Figure 6.4). Depending on the in-lining transformation, it may be possible to map `foo` and `blah` individually to sub-sets of instructions in `foo_n`. If the performance tool allows for basic-block or instruction granularity of instrumentation, and if all of `foo_n` code can be attributed to either `foo` or `blah`, then the resulting resource mappings are 1-to-N. The example in Figure 6.4, shows a 1-to-2 mapping from `foo` to two blocks of `foo_n` indicated by address ranges: *f*:`{foo}→{foo_n/addr00:07, foo_n/addr20:40}`. Requests for performance data for `foo` are mapped to instrumentation of its corresponding two blocks of `foo_n` code. Data collected for `foo_n/addr00:07` and `foo_n/addr20:40` are mapped back and aggregated into a single value that is viewed in terms of `foo`.

Sometimes native code cannot be cleanly divided into blocks attributable to the individual byte-code methods from which it is compiled; byte-code from multiple methods can be mingled in the resulting native code. Two examples are: when instrumentation is available only at a function-level granularity; and when instrumentation is available at a finer level of granularity, but the run-time compiler produces optimized blocks of native code that do not clearly come from an individual method's byte-code.

In the case of function-level granularity of instrumentation, the tool only supports resource mappings at the function and method levels. Mingled code results from run-time compilations of two or more methods into one native code function. For example, if method `foo` is compiled at run-time into method `foo_n` with method `blah` code in-lined, then `foo`, and `blah` both map to function `foo_n`: *f*:`{foo,blah}→{foo_n}` (top of Figure 6.5).

In the second case, mingled code can result when code from two or more methods are optimized by the run-time compiler in such a way that the resulting native code cannot be attributed to individual methods. For example, if methods `foo` and `blah` are compiled into method `foo_n`,

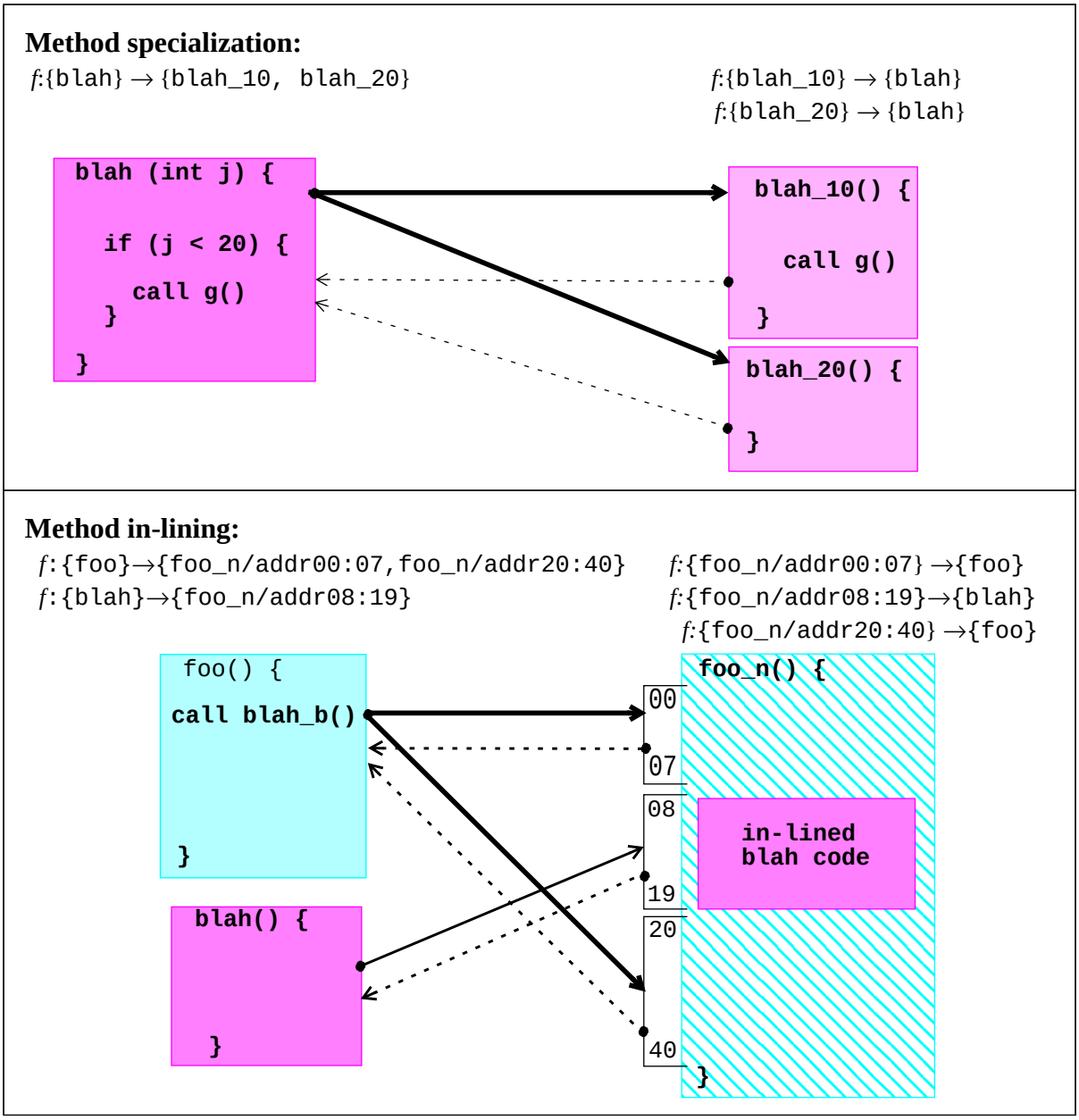

Figure 6.4 1-to-N mappings resulting from method in-lining and specialization: *The top figure is an example of specialization: two specialized versions of method* `blah` *are created for specific values of* `blah`*'s parameter* `j` *(method* `blah` *maps to {*`blah_10` *and* `blah_20`*}). In the bottom figure, method* `foo` *with a call to method* `blah`*, and method* `blah` *are compiled to* `foo_n` *with* `blah` *code in-lined. In this example, address ranges of* `foo_n` *can be attributed to* `foo` *and* `blah`*:* `foo` *maps to two separate address ranges in* `foo_n` *(*`addr00:07` and `addr20:40`*).*

in-lined `blah` code may be optimized with the surrounding `foo` code in such a way that it is impossible to determine which `foo_n` code comes from `blah` and which from `foo`. As a result, both `foo` and `blah` map to these mingled blocks of `foo_n` code: *f*:`{foo,blah}→{foo_n/addr20:40}` (bottom of Figure 6.5).

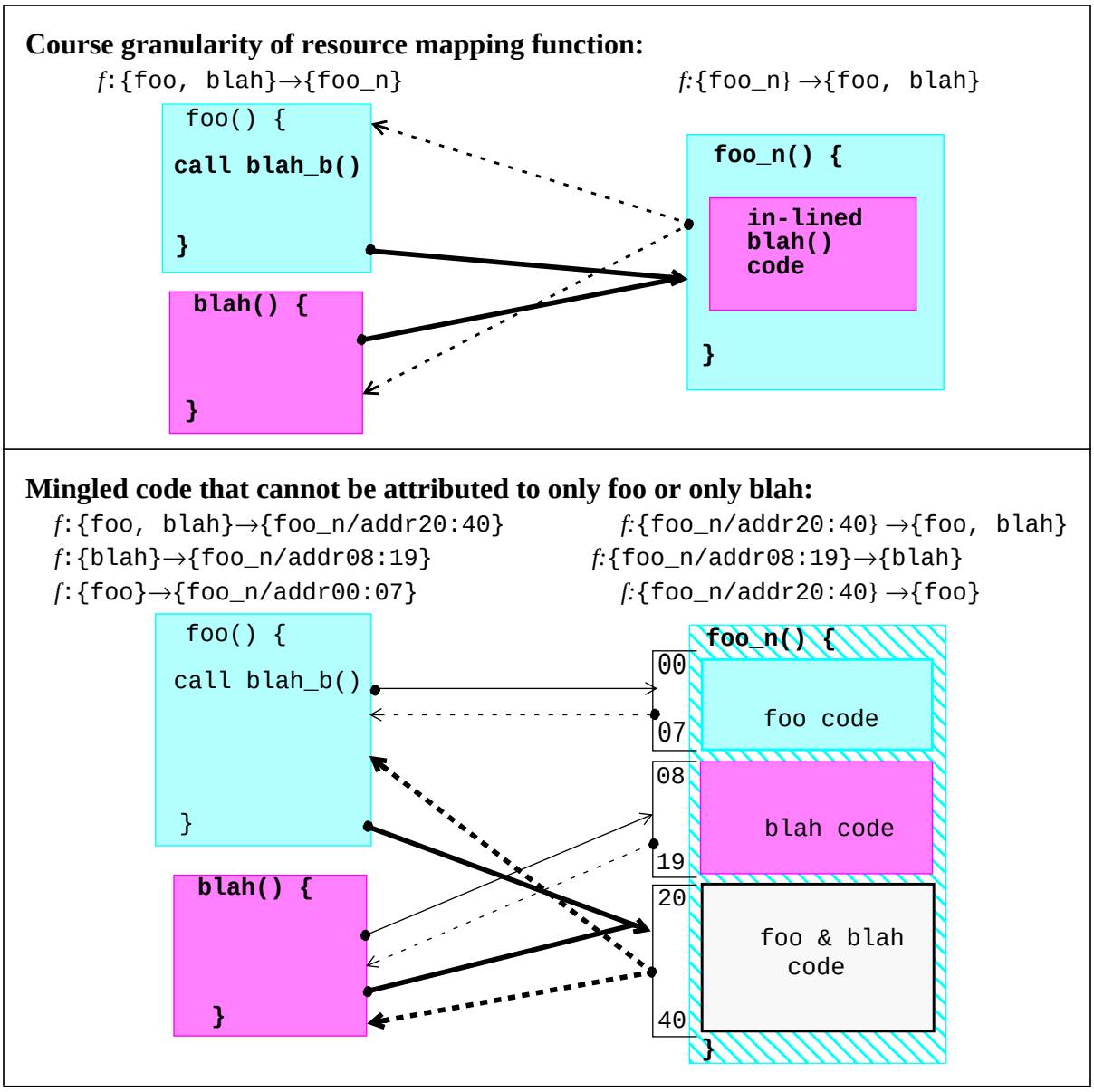

Figure 6.5 N-to-1 mappings resulting from method in-lining with course granularity or mingled code: *The top figure show a 2-to-1 mapping that results from systems that support a course granularity of resource mapping functions (method-level granularity). In this case the set {*`foo, blah`*} maps to {*`foo_n`*}. The bottom figure shows a 2-to-1 mapping that results from mingled code; lines 20 to 40 of* `foo_n` *cannot be attributed to* `foo` *or* `blah` *separately, so the set {*`foo, blah`*} maps to {*`foo_n/addr20:40`*}.*

Mapping performance data and performance data queries is difficult for mingled code; when mingled code is instrumented, the resulting performance data has to be assigned to the byte-code methods from which the mingled code was produced. There are two alternatives for mapping performance data collected mingled native code, either: (1) split the data value between the individual byte-code methods, or (2) create a composite resource node that represents the set of contributing byte-code methods and map the data value back to this composite node.

In the data splitting case, we must determine how to partition the data values collected for the mingled code between the individual byte-code methods from which the native code was compiled. There is no reliable way to determine precisely how to split the cost. Even if we could determine which individual instructions of a mingled block come from which methods, data splitting is still not possible because of the complex execution patterns of modern processors. We might guess based on the methods' relative execution times prior to compilation, or based on the relative code sizes of the methods, but there is no reliable way to split the data.

Another alternative is to map the data back to a composite node representing the set of methods from which the mingled code was compiled. For example, performance data collected for the mingled function `foo_n` is mapped back to the composite node {`foo`, `blah`}. In this case, it is easy to determine how to map performance data values for `foo_n` code back to byte-code views. However, this solution causes problems for data queries in terms of AP byte-code; a user cannot easily ask for performance data in terms of method `foo`, because now part of `foo`'s execution is associated with the synthesized constraint {`foo`, `blah`}; the user is left with the task of determining how much of the data associated with the synthesized node {`foo`, `blah`} comes from method `foo`.

Finally, run-time compilations that combine specialization with method in-lining can result in many-to-many resource mappings. Performance data queries and performance data are mapped between native and byte-code forms of AP code using techniques that combine the mingled code and specialization techniques.

Figure 6.6 summarizes how resource mapping functions are used to map performance data queries and performance data between AP code forms for each possible type of resource mapping function.

6.2 Representing Performance Data

We describe our representation of performance data associated with AP code objects with multiple execution forms. We represent *form-dependent* performance data that are measured for only one execution form of and AP constraint, and we represent *form-independent* performance data that are measured for all execution forms of an AP constraint and viewed in terms of the initial form of the constraint.

6.2.1 Representing Form-Dependent Performance Data

Form-dependent performance data is data associated with only one execution form of an AP constraint. We represent form-dependent data as a metric-focus pair containing AP constraints. The following are examples of form-dependent performance data:

- **objectCreateTime**, **</APCode/Blah.class/foo>:** the amount of object creation overhead in the VM due to objects created in method `foo`'s interpreted execution.
- **objectCreateTime, </APCode/Blah_native/foo>:** the amount of object creation overhead in the VM due to objects created in function `foo`'s direct execution.

Mapping	Example	Mappings performance data queries from byte-code to native code	Mapping performance data from native to byte-code
1-to-1	Byte-code method `foo` is compiled to native code function `foo_n.`	Requests in terms of `foo` map to instrumentation of `foo_n.`	Performance data collected for `foo_n` is mapped back to `foo.`
1-to-M	**Specialization:** Method `blah` is compiled to two specialized native functions `blah_10` and `blah_20` (1-to-2 mapping).	Request in terms of `blah` map to instrumentation of `blah_20` & `blah_10.`	Performance data collected for `blah_20` and `blah_10` is mapped back to `blah` and aggregated (e.g. summed) into a single value.
N-to-1	**In-lining:** Byte-code method `foo` is compiled to `foo_n` with `blah` in-lined (2-to-1 mapping).	Request in terms of `foo` results in instrumentation of parts of `foo_n`: (1) instrument parts of foo_n attributed to only foo and (2) instrument parts of `foo_n` attributed to both `blah` and `foo` if data splitting is used Requests in terms of `blah` are handled in a similar way.	Data collected for `foo_n`: (1) Measures for the `foo` parts are mapped back to `foo` and aggregated into a single value. (2) Measures for the {`foo, blah`} parts are either: (a) mapped back to set {`foo_n, blah_n`}, or (b) split between `foo` and `blah` and aggregated with part (1) data.
N-to-M	**Specialization + In-lining:** byte-code method `foo` and `blah` are compiled into two specialized native code versions (`foo1_n`, `foo2_n`) with `blah` code in-lined (2-to-2 mapping).	Request in terms of `foo` is handled in similar way as the in-lining case, but applied to both `foo1_n` and `foo2_n.`	Performance data collected for `foo1_n` and `foo2_n` are mapped back to `foo` and `blah` in a way similar to the in-lining case, and aggregated into single values.

Figure 6.6 Using resource mapping functions to map performance data. Examples show how performance data queries and performance data values are mapped between AP byte-code to native code. We show examples for each possible mapping function type.

- **objectCreateTime, </APCode/libblah.so/g>:** the amount of object creation overhead in the VM due to objects created in native method `g`.

6.2.2 Representing Form-Independent Performance Data

To represent performance data that is independent of the current execution form of an AP code object, we define a *form-independent flag*, *, that can be associated with metric-focus selections. The flag can be applied to a metric, a focus, or both. It indicates that performance data should be collected for any execution form of the AP byte-code constraints to which it is applied, but viewed in terms of the metric-focus selection.

- Definition 6.2: A *form-independent flag, **, associated with a metric, *M*, a focus, *F*, or a metric-focus pair, (*M,F)*, specifies that data should be collected for all execution forms of AP constraints in the metric, or in the focus, or in both. Given a metric-focus pair, (*M,F*), and a resource mapping function, *f*, for each constraint, *C*, in (*M,F*) produce (*M,F*)* by replacing *C* with (*C* `OR` *f*(*C*)).

For example, if a **Method** instance `foo` is compiled at run-time to **Function** instance `foo_n`, then a request for (`CPUtime, </APCode/foo.c/foo>*`) produces (`CPUtime, </APCode/foo.c/foo OR /APCode/foo_native/foo_n>`). The result is measurement of `CPUtime` for both `foo` and `foo_n`.

When an AP constraint defined in the metric-focus pair *(M, F)** is compiled at run-time, the resulting resource mapping functions are used to map instrumentation to the new execution form of the AP constraint. Performance data collected for the new form of the constraint is mapped back to be view in terms of the original form of *M* and *F*, according to the mapping rules in Figure 6.6. The following are examples of form-independent performance data using the two examples from Figure 6.4:

- **(CPU, </APCode/foo.class/blah>*)** : CPU time when all execution forms of method `blah` are active:

```
  [APCode.constrain(foo.class/blah) OR
   f(APCode.constrain(foo.class/blah)]
  processTime/sec

= [APCode.constrain(foo.class/blah) OR
   APCode.constrain(foo_native/blah_20) OR
   APCode.constrain(foo_native/blah_10)]
  processTime/sec
```

- **(CPU, </APCode/foo.class/foo>*)** : CPU time when all execution forms of method `foo` are active:

```
  [APCode.constrain(foo.class/foo OR
   f(APCode.constrain(foo.class/foo)]
  processTime/sec

= [APCode.constrain(foo.class/foo) OR
   APCode.constrain(foo_native/foo_n/addr00:07) OR
   APCode.constrain(foo_native/foo_n/addr20:40]
  processTime/sec
```

Metrics also can contain `APCode` constraints. For example, `initTime` is a metric that measures the amount of time spent in a Java math library initialization routine: `initTime = [/APCode/Math.class/initialize]wallTime/sec`. The form-independent flag applied to `initTime` will result in data collection for the metric `initTime` even when the math library is compiled to native form. The following are examples of performance data using `initTime` combined with the form-independent flag:

- **(initTime*, /APCode/foo.class/foo):** time spent in any execution form of Math library method `initialize`, when called from only the byte-code form of method `foo`:

```
[APCode.constrain_caller(foo.class/foo)]
AND
[APCode.constrain(Math.class/initialize) OR
 f(APCode.constrain(Math.class/initialize)]
wallTime/sec
```

- **(initTime, /APCode/foo.class/foo)*:** time spent in any execution form of Math library method `initialize` when called from any execution form of method `foo`:

```
[APCode.constrain_caller(foo.class/foo) OR
 f(APCode.constrain_caller(foo.class/foo)]
AND
[APCode.constrain(Math.class/initialize) OR
 f(APCode.constrain(Math.class/initialize)]
wallTime/sec
```

Both of these examples apply the form-independent flag to the metric. In general, the form-independent flag should be applied to any metric that contains `APCode` constraints; since the metric is encoding the `APCode` constraint it makes the most sense to hide the run-time transformation of the `APCode` constraint from the user.

In Figure 6.7, we show an example of form-dependent and form-independent performance data collected from Paradyn-J. Our current implementation of Paradyn-J supports only one-to-one resource mapping functions. In this example we show a form-dependent and a form-independent view of performance data associated with a Java method function that is compiled at run-time to native code.

6.2.3 Representing Transformational Costs

To represent performance measures of run-time compilation costs, we can define metric functions that measure these costs. When combined with foci containing `APcode` constraints, these metrics measure transformational costs for individual AP code objects. Figure 6.8 shows an example of performance data collected from Paradyn-J that measure run-time compilation overhead. The time plot shows the performance metric `compile_time` measured for several Java AP method functions; `compile_time` is a measure of the amount of wall time the VM spends in its run-time compiler code.

6.3 Changes to Paradyn-J to Support Measuring Dynamically Compiled Java Executions

In this section, we present changes to Paradyn-J's implementation to support measuring dynamically compiled Java executions. Ideally, we would have ported Paradyn-J to a real Java dynamic compiler, unfortunately, no source code was available for any of the existing Java dynamic compilers. Instead, we simulated dynamic compilation, and modified Paradyn-J to measure our simulated dynamically compiled executions. We first present our simulation and then the details of Paradyn-J's implementation for measuring the simulated dynamically compiled executions.

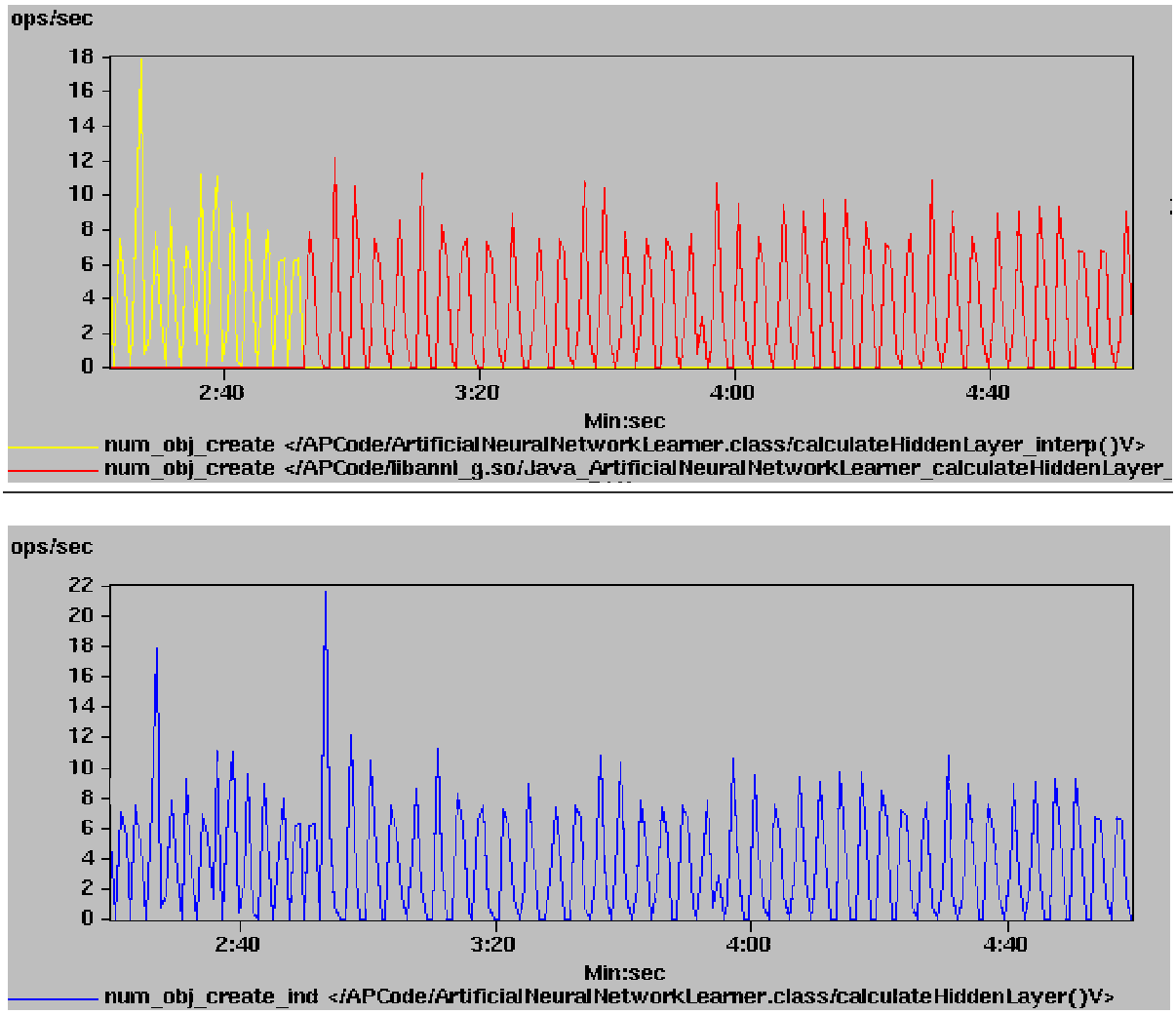

Figure 6.7 Performance data associated with a transformed AP code object. Both time-plot displays show performance data measuring the number of objects created by a Java application method `calculateHiddenLayer()`. The top plot shows two form-dependent measures associated with the byte-code (yellow curve) and native code form (red curve) of the method. The bottom display shows the same performance data represented by one form-independent measure (note: Paradyn-J's current notation for form-independent measures differ from our model, however, the measure in the bottom curve is equivalent to (M,F)* using our model's notation).

6.3.1 Simulating Dynamic Compilation

The goals of our implementation of simulated dynamic compilation, is to use the resulting simulation to demonstrate our model for describing performance data from dynamically compiled executions. To do this, we need to simulate the three main run-time activities in a dynamically compiled execution: (1) interpretation of method byte-code; (2) run-time compilation of some methods; and (3) direct execution of the native form of transformed methods.

We simulate dynamic compilation by modifying a Java application and running it with a Java

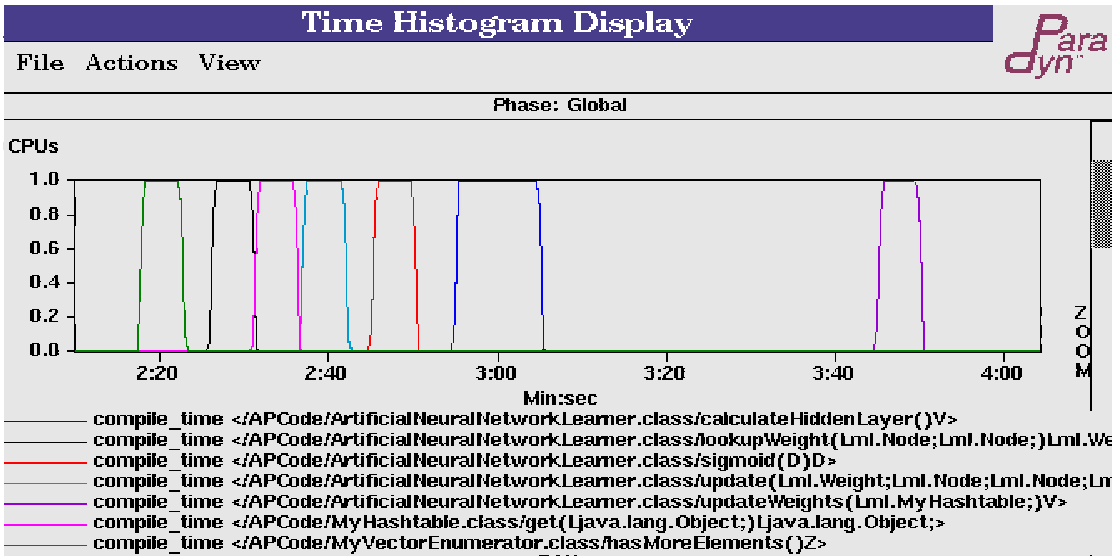

Figure 6.8 Performance Data measuring transformation times of seven methods from a Java neural network application program. *The metric* `compile_time` *measures the amount of time the Java VM spends compiling each method to native code form at run-time.*

interpreter (JDK 1.1.6 running on Solaris 2.6). The VM handles all class loading, exception handling, garbage collection, and object creation. A "dynamically compiled" Java method is replaced with a wrapper method that initially calls a byte-code version of the method that is interpreted by JDK's interpreter. After we reach a threshold (based on number of calls) the wrapper method calls a routine that simulates the method's run-time compilation. The "compiling" routine takes an estimated compiling time as a parameter, and waits for the specified time. For all subsequent calls to the method, the wrapper function calls a native version of the method. The native version is written in C with minimal use of the JNI interface [61]. It is compiled into a shared object that the VM loads at run-time. We approximate each method's compile time by timing ExactVM's run-time compilation of the method. Figure 6.9 shows an example of how we simulate run-time compilation of a Java method.

Our simulation implements the three different execution phases of a run-time compiled method: (1) interpretation, (2) run-time compilation of some methods, and (3) direct execution of the run-time compiled methods. However, our simulation adds extra overhead that would not be present in a real dynamically compiled execution; each wrapper method adds an extra layer of indirection for calls to byte-code and JNI native code versions of the method, and adds more interpreted execution since it is in byte-code form. Also, our native code versions of "compiled" methods use the JNI interface. Real dynamically compiled code does not need to use JNI because the VM controls how the native code is compiled, thus it can ensure that the native code conforms to Java's safety requirements.

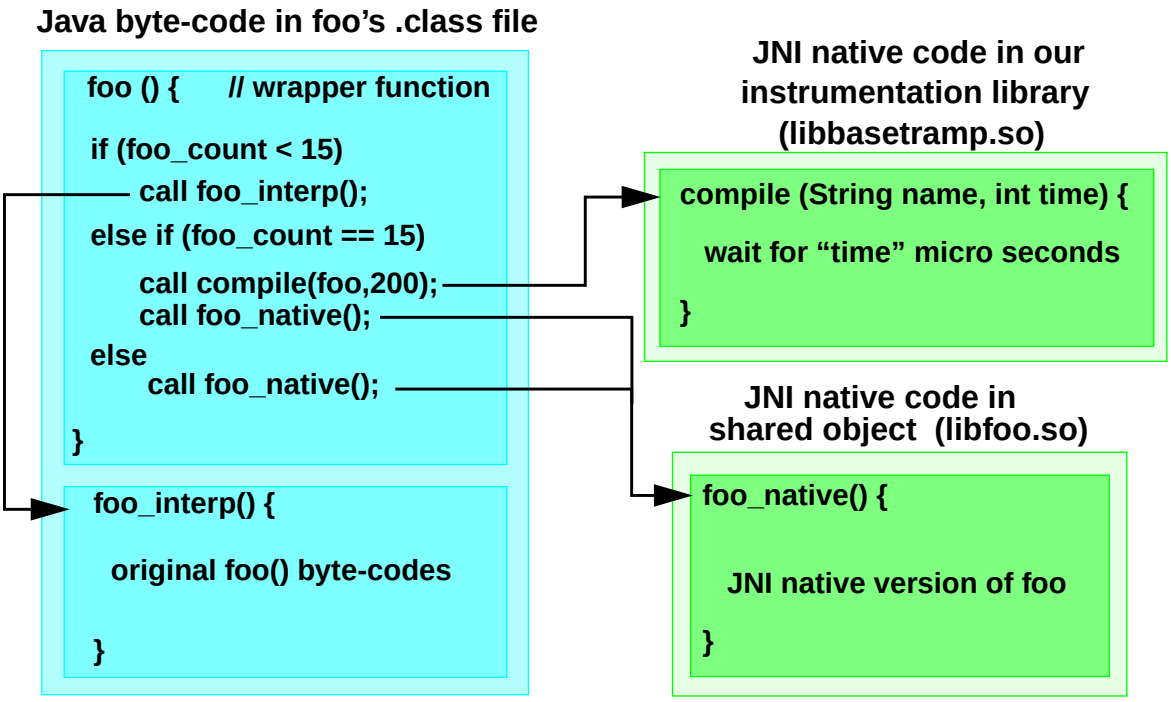

Figure 6.9 Simulation of dynamic compiling method foo. We replace the original method `foo` with a wrapper method (`foo`). For the first 14 executions, the wrapper method calls an interpreted version (`foo_interp`), which contains `foo`'s original byte-code. On the 15th execution, the wrapper calls `compile` that simulates run-time compilation, and all subsequent executions call a JNI native code version of the method (`foo_native`).

6.3.2 Modifications to Paradyn-J

We modified our implementation of Paradyn-J for measuring all-interpreted Java executions, to add support for measuring simulated dynamically compiled executions. We strove to modify Paradyn-J in a way that would be similar to how the tool would be implemented for measuring a real dynamically compiled Java execution.

Paradyn-J instruments the VM's run-time compiling routines to discover the native form of a compiled method, to discover mappings between byte-code and native code forms of a method, and to measure costs associated with the run-time compilation of a method (this is how Paradyn-J would obtain the same information from a real dynamically compiled execution). At runtime, the VM calls `dlopen`() to load the shared objects that contain the JNI native versions of the AP methods and that contain our "compiling" routine. Paradyn-J instruments the VM to catch `dlopen` events. When Paradyn-J detects that the VM has loaded our "compiling" routine, Paradyn-J instruments it. The instrumentation in the compiling routine notifies Paradyn-J when a method is "dynamically compiled". Also, instrumentation at the routine's entry point starts a timer to measure the dynamic compiling time, and instrumentation at its exit point stops the timer measuring the compiling time.

The resource mapping functions are discovered by instrumenting our compiling routine. Currently we support only one-to-one run-time transformations; our compiling routine takes a parameter naming the byte-code method to be compiled, and returns the native code transformations of the method. By instrumenting the entry and exit points of the compiling routine we are able to discover resource mapping functions at run-time. Resource mapping functions are stored with individual resources so that instrumentation requests for byte-code form resources can be mapped to native form resources for data collection, and so that performance data collected in the native code form of a method can be mapped back to be viewed in terms of the byte-code form of the method. Figure 6.7 shows an example of performance data collected from Paradyn-J that uses resource mapping functions to map instrumentation and data between AP constraints contained in a form-independent metric-focus pair.

6.4 Performance Tuning Study of a Dynamically Compiled Java Application

We used Paradyn-J to provide detailed performance data from two Java applications, a neural network application consisting of 15,800 lines of Java source code and 23 class files, and a CPU simulator application consisting of 1,200 lines of code and 11 class files. Using performance data from Paradyn-J, we tuned a method in the neural network application improving the method's interpreted byte-code execution by 11% and its native code execution by 10%, and improving overall performance of the application by 10% when run under ExactVM. We profile the CPU simulator application to further show how we can obtain key performance data from a dynamically compiled execution.

For the neural network program, we picked good candidate methods to "dynamically compile" by using Paradyn-J to measure an all-interpreted execution of the application; we picked the seven application methods that accounted for most of the execution time. We wrote JNI native versions and wrapper functions for each of these methods. We first demonstrate that Paradyn-J can associate performance data with AP methods in their byte-code and native code forms, and with the runtime compilation of AP methods. Figure 6.10 shows a performance visualization from Paradyn-J. The visualization is a time plot showing the fraction of CPUtime per second for the byte-code (in blue) and native (in red) forms of the `updateWeights` AP method, showing that `updateWeights` benefits from dynamic compilation. Figure 6.11 is a table visualization that shows performance measures of total CPUTime (middle column), and total number of calls (right column) associated with the byte-code (top row) and native (middle row) forms of `updateWeights`, and compiling time (left column) associated with the method's wrapper function (0.174 seconds). The visualization shows data taken part way through the application's execution. At the point when this was taken, the procedure calls measure shows that the byte-code version is called 15 times for a total of 0.478 seconds before it is "dynamically compiled", and the native code version has executed 54 times for a total of 0.584 seconds. The implication of this data is that at this point in the execution, `updateWeights` has already benefited from being compiled at runtime; if

the method was not "dynamically compiled", and instead was interpreted for each of the 69 calls, then the total execution time would be 2.2 seconds (69 calls × 0.031 seconds/call). The total execution time for the method's "dynamically compiled" execution is 1.2 seconds (0.478 seconds of interpreted execution + 0.174 seconds of compilation + 0.584 seconds of native execution).

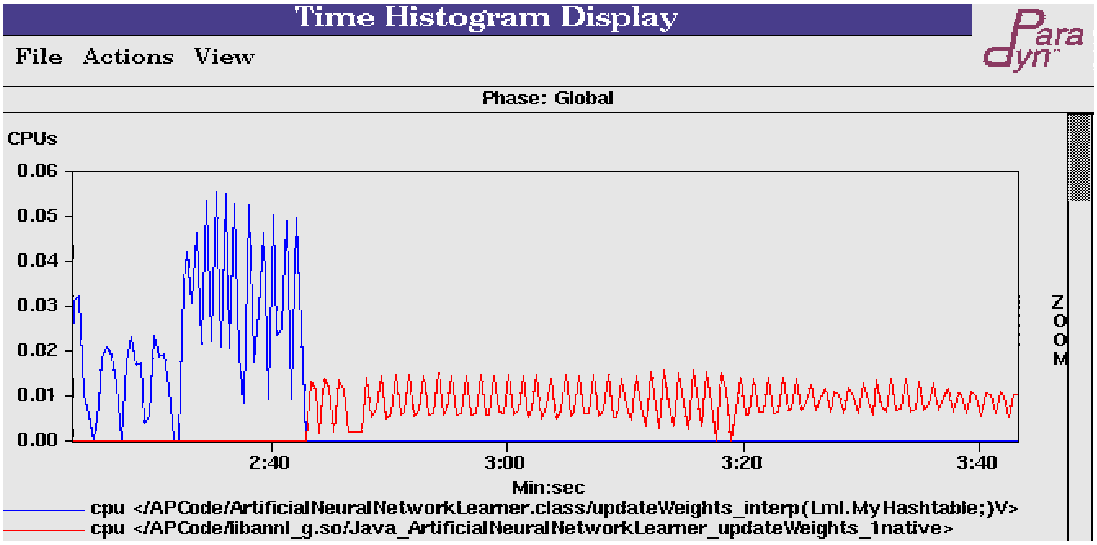

Figure 6.10 Performance data for the `updateWeights` method from the dynamically compiled neural network Java application. *The time plot visualization shows the fraction of CPUtime/second for the native (red) and byte-code (blue) form of the method.*

Table Visualization

File Actions View

Phase: Global

	compile_time CPUs_seconds	cpu CPUs_seconds	procedure_calls ops
updateWeights_interp(Lml.MyHashtable;)V		0.478	15
Java_ArtificialNeuralNetworkLearner_updateWeights_1native		0.584	54
updateWeights(Lml.MyHashtable;)V	0.174		69

Figure 6.11 Performance data for the `updateWeights` method from the dynamically compiled neural network Java application. *The table shows the performance measures total* `CPUTime` *(second column) and number of calls (third column), for both the byte-code (top row), and native (middle row) forms, and compile time (first column) associated w/the wrapper (bottom row).*

We next demonstrate how performance data from Paradyn-J can explicitly represent VM costs associated with byte-code and native code forms of a method. We measured the number of object creates in each of our "dynamically compiled" methods. In Figure 6.12, the visualization shows a method (`calculateHiddenLayer`) that accounts for most of the object creates. The visualization

shows data taken part way through the application's execution. In its byte-code form (top row), the method is called 15 times, creates 158 objects, and accumulates 3.96 seconds of CPU time. After it is called 15 times, it is compiled at runtime, and its native code form (bottom row) is called 50 times, creates 600 objects, and accumulates 20.8 seconds of CPU time[1]. Its native form execution is more expensive (at 416 ms per execution) than its interpreted execution (at 264 ms per execution). These performance data tell the Java AP developer that in both its byte-code and native code form, `calculateHiddenLayer` creates a lot of objects. At least part of the reason why it runs so slowly has to do with the VM overhead associated with these object creates. One way to improve the method's performance is to try to reduce the number of objects created in its execution. We examined the method's Java source code, and discovered that a temporary object was being created in a `while` loop. This temporary object had the same value each time it was created and used inside the loop. We modified the method to hoist the temporary object creation outside the loop. The table in Figure 6.13 shows total CPUtime and object creates of the modified version of `calculateHiddenLayer` (the data was taken partway through the application's execution). As a result of this change, we were able to reduce the number of object creates by 85% in the byte-code version (23 vs. 158 creates), and 75% in the native code version (150 vs. 600 creates). The CPU time spent interpreting the method's byte-code form improved by 11% (3.53 vs. 3.96 seconds), and the CPUtime executing the method's native code form improved by 10% (18.7 vs. 20.8 seconds).

Table Visualization

File Actions View

Phase: Global

	cpu_inclusive CPUs_seconds	num_obj_create ops	procedure_calls ops
calculateHiddenLayer_interp()V	3.9614	158	15
Java_ArtificialNeuralNetworkLearner_calculateHiddenLayer_1native	20.762	600	50

Figure 6.12 Performance data for method `calculateHiddenLayer`**.** *The total CPU time (first column), total number of object creates (second column), and total number of calls (third column) to the byte-code (top row) and native code (bottom row) forms of the method.*

Next, we wanted to see how well our tuning based on a simulated dynamically compiled execution translates to a real dynamically compiled execution. We performed the same tuning changes to the original version of the Java application (without our modifications to simulate dynamic compilation), and measured its execution time when run under ExactVM. The overall

1. Each time the method is called, the number of object creates can vary due to changes in the application's data structures.

Table Visualization

File Actions View

Phase: Global

	cpu_inclusive CPUs_seconds	num_obj_create ops	procedure_calls ops
calculateHiddenLayer_interp()V	3.53	23	15
Java_ArtificialNeuralNetworkLearner_calculateHiddenLayer_1native	18.7	150	50

Figure 6.13 Performance data for method `calculateHiddenLayer` after removing some object creates. *This table shows that the total CPUtime for both the native and byte-code forms of the method is reduced as a result of reducing the number of object creates.*

execution time improved by 10% when run by ExactVM with dynamic compiling, and by 6% when run by ExactVM with dynamic compiling disabled (Table 6.14). These results imply that ExactVM's interactions with AP native and byte-codes due to handling object creates account for a larger percent of the application's execution time (compared to our "dynamic compiler"). ExactVM has improvements over JDK 1.1.6 to reduce garbage collection, method call and object access times, and it does not have any of the JNI interactions with the VM that our native forms of methods have with the VM. Therefore, it is reasonable to conclude that object creates account for a larger percentage of the VM overhead in ExactVM executions. As a result, our tuned application achieves a higher percentage of total execution time improvement when run under ExactVM than when run by our simulated dynamic compiler.

In this study, we limited our options for performance tuning to the seven methods for which we simulated dynamic compilation. However, there are close to 1,000 methods in the application's execution. If this was a real dynamically compiled execution, then all of the 1,000 methods would be available for performance tuning. Performance data from our tool that can measure VM overhead associated with the byte-code and native code form of a method, help a program developer focus in on those methods to tune, and gives an indication of how to tune the method to improve its performance.

	Original	Tuned	Change
Dynamic Comp.	21.09	18.97	10%
All-Interpreted	190.83	179.90	6%

Figure 6.14 Total execution times under ExactVM for the original and the tuned versions of the neural network program. *We improve the performance by 10% with dynamic compiling, and by 6% with dynamic compiling disabled (all-interpreted).*

Our performance data can help a programmer identify why their application does not benefit

from run-time compilation. For example, in Chapter 5 we demonstrated cases where if we had performance data describing specific VM costs and I/O costs associated with a method's interpreted byte-code and directly executed native code, then we could more easily determine how to tune the method to improve its performance.

Case 1: object creates		Case 2: I/O intensive			Case 3: small functions		
Measurement	**Byte-code**	**Measure-ment**	**Native**	**Byte-code**	**Measure-ment**	**Native**	**Byte-code**
Total CPU seconds	2.3515 s	***Total I/O seconds***	5.6493 s	0.36658 s	***CPU seconds***	4.9 µs	6.7 µs
Object Creation Overhead seconds	1.5730 s	***Total CPU seconds***	0.0050 s	0.04403 s	***MethodCall Time***		2.5 µs

Figure 6.15 Performance data from the CPU Simulation AP. *Performance measurements of methods in the AP that have performance characteristics similar to the three test cases from Chapter 5.*

In our second study, using the CPU simulator application, we show additional examples of how Paradyn-J can provide the type of detailed performance measures that we discovered would be useful in Chapter 5; we picked methods to "dynamically compile" based on the three cases we examined in Chapter 5. For the first case (native code with a lot of VM interaction), we picked a method that created several String objects. For the second case (methods whose execution is not dominated by interpreting byte-code), we picked a method that performed a lot of I/O. For the third case (small byte-code methods), we picked a method consisting of three byte-code instructions that simply returned the value of a data member. In Table 6.15, we show performance data from Paradyn-J's measurement of each of the three methods.

For case 1, VM object creation overhead account for more than half of the method's execution time (1.57 out of 2.35 seconds); this tells the AP developer that one way to make this method run faster is to try to reduce this VM overhead by removing some object creates from the method's execution.

In the second case, a method that performs a lot of I/O, our tool can represent performance data showing the amount of CPU seconds and I/O seconds in the interpreted byte-code and directly executed native code form of the method (a total of 5.65 seconds of I/O time and negligible CPU time in the native code form, and a total of 0.37 seconds of I/O time and 0.044 seconds of CPU time in the byte-code form)[1]. The performance data tell an AP developer to focus on reducing

1. The I/O time for the native code is much larger than that of the byte-code because the native code of the method is called more frequently than the 15 calls to the interpreted byte-code form of the method. We are representing these numbers as total rather than per call numbers because each call to the method writes a different number of bytes; they are not directly comparable on a per call basis.

the I/O costs since they account for the largest fraction of the method's execution time (almost 100% of the native code's execution, and 90% of the interpreted byte-code's execution is due to I/O costs).

In the third case, small method functions with a few simple byte-code instructions, our performance data represent CPU times for both the byte-code and native code form of the method. The data provide us with some explanation of why this method benefits from being dynamically compiled; the fraction of CPU time for the native code version of the method is slightly better than for the byte-code version (4.9 μs to 6.7 μs per call), however, the added method call overhead for interpreting the byte-code call instruction (an additional 2.5 μs for every 6.7 μs of interpreting byte-code) makes interpreted execution almost twice as expensive as native execution. If this had been an all-interpreted execution, then the performance data for the interpreted byte-code form of the method indicates that interpreting method call instructions is an expensive VM activity. Therefore, one way to make this method run faster on an interpreter VM is to reduce the number of method calls in the execution. The performance data from these three methods describe the detailed behaviors needed by AP developers to tune their dynamically compiled applications.

6.5 Our Performance Data and VM Developers

The same type of performance data used by an AP developer can also be used by a VM developer to tune the VM. For example, by characterizing byte-code sequences that do not benefit much from dynamic compilation (like methods with calls to I/O routines and simple control flow graphs), the VM could identify AP methods with similar byte-code sequences and exclude them from consideration for run-time compilation. Similarly, performance data showing that certain types of methods may be good candidates for compiling, can be used by the VM to recognize these methods, and compile them right away (ExactVM does something like this for the case of methods containing loops). The data can also point to ways that the compiler can be tuned to produce better native code. For example, performance measures indicating that VM method call overhead is expensive can be used to tune the compiler to aggressively in-line methods (the high cost of interpreting method call instructions is the reason why HotSpot is designed to aggressively in-line methods).The VM also could use performance information about specific interactions between the VM and the native code (e.g., object creation overhead) to try to reduce some of these expensive VM interactions or to tune the VM routines that are responsible for these interactions (e.g., the VM routines involved in object creation).

Detailed performance data, collected at runtime, could be used to drive the VM's runtime compiling heuristics. For example, the VM could measure I/O and CPU time for a method the first time it is interpreted. If the method is dominated by I/O time, then exclude it as a candidate for compiling (and stop profiling it). There have been several efforts to incorporate detailed runtime information into compilers to produce better optimized versions of code and/or to drive runtime compiling heuristics [74, 29, 14, 3] (these are all for languages other than Java).

6.6 Conclusions

In this chapter we presented a representational model for describing performance data from AP's with multiple execution forms. Our model is a guide for what to build in a performance tool for measuring programs with multiple execution forms. We presented a proof-of-concept implementation of our model: modifications to Paradyn-J to add support for measuring a simulation of dynamically compiled Java programs. In a performance tuning study of a Java application program, we demonstrated how the type of performance data that can easily be described by performance tools based on our model, allows a program developer to answer questions of how to tune the AP to improve its performance. For example, we showed how performance data describing a specific VM costs (object creation overhead) associated with the interpreted byte-code and the directly executed native code of a transformed AP method function, lead us to determine how to tune the AP (by removing some object creates), and where to focus our tuning efforts (remove object creates in AP method `calculateHiddenLayer`). As a result, we were able to improve the method's performance by 10%. We also described how performance data from Paradyn-J can answer questions about the virtual machine's execution, and we discussed how a dynamic compiler VM could incorporate performance data into its run-time compiling heuristics.

Chapter 7

Lessons Learned from Paradyn-J's Implementation

We learned many lessons from implementing Paradyn-J. In particular, some of our design choices were difficult to implement and our byte-code instrumentation is expensive. In this chapter, we discuss our implementation, and characterize ways in which a performance tool based on our model must be implemented to provide performance data that describe specific VM-AP costs in terms of the multiple execution forms of AP code. We also discuss alternative ways to implement a tool based on our model that avoid some of the costs of Paradyn-J's instrumentation technology, or that make Paradyn-J easier to port to other virtual machines. In particular, we discuss Sun's new profiling tool interface JVMPI. Our work points to places where JVMPI needs to be expanded to provide the type of performance data that we have demonstrated is critical to understanding the performance of Java executions; currently JVMPI provides no way to obtain explicit VM performance information, nor does it provide a way to obtain information about arbitrary interactions between VM and AP program resources.

7.1 Issues Related to the Current Implementation of Paradyn-J

Paradyn-J was complicated to implement, in part because it uses detailed knowledge of the internals of the Java VM. As a result, porting Paradyn-J to a new VM requires a non-trivial effort. Also, our decision to instrument unmodified Java application .class files resulted in an instrumentation technology that is difficult to port to new VM's and that can greatly perturb an AP's execution. Ideally, we would like Paradyn-J to be easily ported to new VMs, and we would like an instrumentation technology that results in a minimal amount of perturbation. However, some of these complications are unavoidable side effects of implementing a tool based on our model.

One difficulty with Paradyn-J's implementation is that it requires knowledge of the internals of the VM, and as a result, requires a non-trivial effort to port to different VMs. Paradyn-J needs knowledge of the VM: (1) to know which VM code to instrument to measure certain VM activities, (2) to discover when the VM loads Java AP .class files so that it can create AP code resources, (3)

to discover resource mapping functions and native code forms of AP code that is compiled at run-time, and (4) to determine when it is safe to insert instrumentation into Java AP byte-codes at run-time.

Paradyn-J parses the Java VM executable file and shared object files to create VM code resources. However, Paradyn-J also needs to know details about particular VM functions to create VM-specific metrics and to identify certain VM run-time events. To create VM-specific methods that measure VM overhead like garbage collection time, object creation time, and method call time, we had to examine the VM interpreter code to discover how the VM interprets method call and object creation AP byte-codes, and how garbage collection is implemented in the VM. After examining VM source code we were able to find the VM routines responsible for handling these run-time events, and then to create VM-specific metric functions that measure their performance.

Java executions are extremely dynamic. AP classes can be loaded and unloaded and AP methods can be translated from byte-code to native code at any point in the execution. To implement Paradyn-J, we had to find the parts of the VM code responsible for loading AP .class files. Paradyn-J needs to instrument VM class loading routines to identify when a Java .class file has been loaded, and needs to instrument the run-time compilation routine to identify when a Java method function is dynamically compiled at run-time. These routines differ between VM implementations, so porting Paradyn-J to a new VM requires identifying which VM routines to instrument and how to instrument them to obtain AP resource and mapping information.

Instrumentation code in the VM class file loading routines notifies Paradyn-J of the location of the internal data structures in the VM; these data structures identify the new AP code resources for the loaded class. Because Paradyn-J parses AP byte-codes after they have been loaded by the VM, we had to examine VM source code to find the data structures used by the VM to store AP class file information. Paradyn-J was designed to parse the VM's internal data structures to find the name, size and location of the methods, and to find the class's constantpool. Paradyn-J creates new resources for the class and the methods using information that it obtains from VM data structures, parses each method's byte-code instructions to find instrumentation points, and parses the class' constantpool so that it can be modified with information to resolve the method call byte-code instruction used by our byte-code instrumentation.

Paradyn-J instruments the simulated run-time compiling routine to obtain resource mapping functions and to find and parse the resulting native code form of the compiled methods. If Paradyn-J was ported to a real dynamic compiler, then we would have to examine its run-time compiling routine to determine how to re-write Paradyn-J to obtain mapping information and to locate the native code form of compiled methods.

Finally, we need to know the internals of the Java VM to obtain the AP's execution state at run-time. The execution state of the AP must be examined to determine if it is safe to insert Java byte-code instrumentation into the process. For a tool that inserts instrumentation code into Java

method byte-codes at any point in the execution, it is not always safe to insert instrumentation; for example, if the VM is interpreting AP instructions within the instrumentation point, then Paradyn-J cannot safely overwrite the method's instructions with a jump to instrumentation code. To implement a run-time check for safety, Paradyn-J examines data structures used by the VM to keep track of the AP's execution state to determine when it is safe to insert instrumentation code. To handle delaying unsafe instrumentation, Paradyn-J inserts special instrumentation code into a method further down in the execution stack. The data structures for storing execution state differ from VM to VM, thus making this part of Paradyn-J less portable to new VMs.

If we had ported Paradyn-J to a real dynamic compiler VM, then we would have to deal with one more VM-specific problem: how to handle instrumented byte-codes that are about to be transformed by a dynamic compiler. One option is to let the VM compile the byte-code instrumentation along with the byte-code instructions of the AP method. However, this solution is not ideal because there is no guarantee that the VM will produce transformed instrumentation code that is measuring the same thing as the byte-code instrumentation (the compiler could re-order instrumentation code and method code instructions, or could optimize away some instrumentation code). A better option is to remove byte-code instrumentation from the method just prior to compilation, let the VM compile the method, and then generate equivalent native code instrumentation and insert it into the native form of the method, requiring that Paradyn-J interact with the VM immediately before and after compilation of a method to un-instrument Java method byte-codes before run-time compilation, and to re-instrument the method's native code form after run-time compilation.

Paradyn-J is a complete implementation of a performance tool based on our model for describing performance data from interpreted, JIT compiled, and dynamically compiled program executions. All parts of the VM and the AP are made visible by the tool, and as a result, most VM-AP interactions can be represented. A nice feature of Paradyn-J is its ability to instrument unmodified VM binaries and unmodified AP .class files; this means that Paradyn-J can measure VM and AP code without requiring that special instrumented versions of the programs be built by re-compiling with an instrumentation library or by modifying the program source code. Also, because Paradyn-J does not need application source to instrument either the VM or the AP, it can instrument system libraries and Java Class Libraries just like any other code in the application. However, Paradyn-J's Transformational Instrumentation is expensive (as discussed in Chapter 4), and Paradyn-J's implementation is VM dependent; porting Paradyn-J to a new VM requires a detailed understanding of the internal structure of the VM program.

7.2 Alternative Ways to Implement a Tool Based on Our Model

We discuss some alternative ways in which a tool based on our model could be implemented. We focus on issues associated with perturbation caused by instrumentation code, ease of porting the tool to a new VM, and how complete a model can be built using the proposed alternative

approach. In general there is a trade-off between how much of the model can be implemented and instrumentation perturbation or portability of the tool.

7.2.1 Requirements for Implementing Our Model

Because our model describes arbitrary, general VM-AP interactions from interpreted, JIT and dynamically compiled executions, and because it represents transformational mapping functions between different forms of AP code objects that are compiled at run-time, the internals of the VM have to be accessible.

The Java VM implements several abstractions that a performance tool needs to see. For example, Java threads and synchronization primitives are AP resources that a performance tool needs to identify. A tool also needs to identify when run-time events involving these resources are occurring in the execution. One way in which this can be achieved is if the VM exports some type of debugging interface that the performance tool can use to obtain this information at run-time. This is similar to how an operating system implements a debugging interface (e.g., System V UNIX's /proc file system [2]) and provides system calls that can be used to obtain information about its abstractions (e.g., process time). The VM could implement a similar interface to export information about its abstractions (such as threads, synchronization, and memory management).

If the VM does not explicitly export it abstractions through such an interface, then the tool has to make parts of the VM visible to correctly measure performance data associated with the VM's abstractions. Paradyn-J had to be implemented in this way because version 1.1.6 of JDK's VM does not export a debugging interface. Performance tools for measuring applications that use user level thread libraries [72] have to solve a similar problem of making thread creation, termination, and context switching visible to the tool, so that the tool can correctly measure timing information associated with a thread's execution.

A complete implementation of a tool based on our model is not possible without either the VM exporting its abstractions via an interface or the performance tool obtaining this information by being explicitly ported to different versions of a VM.

7.2.2 Implement as a Special Version of VM

One way in which a tool based on our model can be built is as a special version of the VM that exports performance information about itself and about the AP it executes. The VM can be built so that it correctly measures timing values for AP code in the presence of thread context switches and run-time compilation of AP code. It could also easily measure specific VM costs associated with its execution of the AP. An implementation that is a special version of the VM has the potential to limit the amount of instrumentation perturbation because no AP byte-codes need to be instrumented. Instead, VM code is re-written to export and measure its execution of the AP. However, there are portability issues associated with this approach. A tool implemented in this way

would be more difficult to port to a new VM than Paradyn-J because all the special modifications to the VM need to be ported to the new VM.

7.2.3 Using the JVMPI Interface

Another way in which a tool based on our model can be implemented is by using a debugging interface exported by the VM. This option is newly available with the Java 2 Platform release of JDK that includes a profiling interface JVMPI [62] (in Chapter 2, we discussed JVMPI's implementation). The benefit of this approach is that the tool is completely portable–the tool can run on any VM that implements this interface. However, there are some problems with the current implementation of JVMPI that result in only parts of our model being implementable using JVMPI.

The JVMPI interface provides functions that a tool can use to query about the state of an AP's execution, and it provides a VM event callback mechanism to notify the tool of certain run-time events. The interface has some nice functionality in terms of obtaining thread level and method level CPU time values. Also, JVMPI provides notification of run-time events such as object creates, garbage collection, monitor entry and exit, class file loading, and JIT compiled method loading. Using JVMPI, method, thread, and object resources can be discovered at run-time, count data can be collected for object creates, method calls, and thread creates, and time data can be obtained at the exit points of methods, garbage collection activities and monitor synchronization events. However, JVMPI hides most of the VM from the profiler agent and, as a result, specific VM costs cannot be measured for an AP's execution.

The biggest shortcoming of JVMPI is that it does not export the necessary events to obtain performance measures associated with specific VM activities (such as method call and object creation overhead). Also, JVMPI provides only coarse grained event notification; a profiler agent can be notified of all method entry events or of none. Another shortcoming is that there are no dynamic compiler events. As a result, transformation times and resource mapping functions cannot be obtained through this interface. Finally, this interface is of little use to a VM developer because very little of the VM is exported via JVMPI.

The JVMPI interface has the potential to be useful for implementing parts of our model useful to an AP developer. Our work can act as a guide to the JVMPI developer on how to expand the JVMPI interface to provide some types of data that we found are critical to understanding the performance of Java executions.

7.2.4 Changes to JVMPI for a more Complete Implementation

Currently JVMPI is limited by two factors: (1) much of the Java VM is not made visible by this interface, and (2) JVMPI allows for only coarse-grained profiling decisions. To implement our model completely, all of the VM must be made visible by JVMPI so that any VM-AP interaction

can be measured. If at least part of the VM was exposed, JVMPI could make available some measures of VM activities. For example, not only should we be able to measure the time spent in method function `foo`, but we also should be able to obtain measures for how much of that time was due to the VM handling method calls, object creates, thread context switches, class file loading, or byte-code verification. With the current interface, we can be notified that an object creation or class file loading event has occurred in the execution, but not how much time these events took, nor how much of an AP method's execution time is due to these events. Also, JVMPI should provide functions for examining all of the virtual architecture implemented by the VM. For example, functions to examine values on a method's argument stack or operand stack, and to access the virtual registers, should be provided by JVMPI. With this type of access, a performance tool can examine the run-time state of the AP at any point in its execution, can obtain operand stack values from methods on the execution stack, or can change the VM's register values to jump to special instrumentation code.

Run-time perturbation is also a problem with JVMPI. All JVMPI calls and callbacks use the JNI interface. JNI is expensive, in part, because of the safety restrictions specified by the Java language [21]. For example, it takes many JNI calls to access a Java object, to access an object's field, or to call a method function. JVMPI allows for only coarse-grained profiling decisions. For example, either the profiler agent can be notified of every method entry event in the execution or of none. As a result, there will often be more run-time perturbation by the profiler agent than is necessary to collect the performance data that the tool user wants. One way to reduce some of these costs is to allow for a finer granularity of event notification. For example, if the profiling tool can specify that it is interested only in method entry and exit events for method `foo`, or for thread `tid_1` executing method `foo`, then this would reduce the amount of event notification in the execution, which in turn would reduce the amount of profiler agent code executed. Another way to reduce some of the costs is to allow the profiler agent code to be written in Java. If the Java VM is a dynamic compiler, then it can produce native code that does not have to go through the JNI interface and, as a result, the profiler agent code will be more efficient.

Because JVMPI provides a fixed set of interface functions for obtaining information about an AP's execution, our model of describing performance data for arbitrary, general VM-AP interactions cannot be completely implemented using JVMPI. However, with changes to JVMPI, particularly to make more of the VM visible to the profiling agent, some of the measures that we found useful in our performance tuning studies could be obtainable through this interface.

7.3 Conclusions

We discussed some of Paradyn-J's implementational lessons, and problems associated with the portability of Paradyn-J. Although Paradyn-J is more portable than some other ways of implementing a tool based on our model, porting Paradyn-J to a new VM requires detailed knowledge of the workings of the VM and the underlying OS/architecture. We discussed some alternative meth-

ods of implementing a tool based on our model, and examined trade-offs in portability and completeness of the implementation and portability and perturbation costs. To be truly portable, our tool would have to be built using only a Java API like the new JVMPI interface. However, the current implementation of JVMPI severely limits how much of the VM we can see, and thus how much of our model can be implemented.

Chapter 8

Conclusion

8.1 Thesis Summary

As Java is increasingly being used for large, long running, complex applications, there is a greater need for performance measurement tools for interpreted, JIT compiled and dynamically compiled program executions. In this dissertation we presented a solution to the difficulties associated with performance measurement of these types of executions, namely that there is an the interdependence between the virtual machine and the application's execution, and that applications can have multiple executions forms. Our solution is a representational model for describing performance data from interpreted, JIT compiled and dynamically compiled program executions that explicitly represents both the VM and the AP program, that describes performance data associated with any VM-AP interaction in the execution, that describes performance data associated with the different execution forms of AP code, that represents the relationships between different forms of AP code, and that describes performance data in terms of both the VM-developer's and the AP developer's view of the execution. We presented a performance tool, Paradyn-J, that is an implementation of our model for measuring interpreted and dynamically compiled Java executions. Finally, we demonstrated the usefulness of our model by using performance data from Paradyn-J to improve the performance of an all-interpreted Java application and a dynamically compiled Java application.

This dissertation presented new methods for describing performance data from interpreted, JIT compiled, and dynamically compiled executions. Our representational model allows for a concrete description of behaviors in interpreted, JIT and dynamically compiled executions, and it is a reference point for what is needed to implement a performance tool for measuring these types of executions. An implementation of our model can answer performance questions about specific interactions between the VM and the AP; for example, we can describe VM costs such as object creation overhead associated with an AP's interpreted byte-code and directly executed native code form. Our model represents performance data in a language that both an application program

developer and a virtual machine developer can understand; we presented VM-specific metrics that encode information about the VM in a language that matches the AP developer's view of the execution. Also, the model describes performance data in terms of different execution forms of an application program object; using resource mapping functions we find and measure the native code form of AP code that is compiled at run-time. Finally, our model describes run-time transformational costs associated with dynamically compiled AP code, and correlates performance data collected for one form of an AP object with other forms of the same object; using the form-independent flag, performance data can be collected for any execution form of AP code constraints.

We discussed the implementation of Paradyn-J, a prototype performance tool for measuring interpreted and dynamically compiled Java executions that is based on our model. In two performance tuning studies using Paradyn-J, we demonstrated how performance data that is described in terms of our model provides information that is critical to understanding the performance of interpreted and dynamically compiled executions. In the all-interpreted performance tuning study, performance data from Paradyn-J identified expensive Java VM activities (method call and object creation overheads), and associated these overheads with constrained parts of the Java application. With these data, we were easily able to determine how to tune the Java application to improve its performance by more than a factor of three. In the performance tuning study of a dynamically compiled Java neural network application, we showed that Paradyn-J provides performance measures of VM object creation overhead associated with the byte-code and native code forms of the dynamically compiled methods. These data allowed us easily to determine how to tune one of the dynamically compiled methods to improve its performance by 10%.

Finally, we discussed some issues associated with Paradyn-J's implementation. In particular, we discussed the necessary features for implementing a tool based our model. Results of our work point to places were JDK's new profiling interface, JVMPI, should be expanded to provide the type of performance data that we found to be useful in our performance tuning studies.

8.2 Future Directions

There are two main directions for future work: (1) extensions to Paradyn-J for a more complete implementation of our model and for measuring parallel and distributed Java executions, and (2) performance measurement studies from the VM-developer's view, in particular, examining how performance data can be used to tune a dynamic compiler VM.

The current version of Paradyn-J only exports a Code view of the AP. One place for extending Paradyn-J is to add support for AP thread and synchronization resource hierarchies. Much of the AP thread support can be leveraged off of work being done to support thread level profiling in Paradyn [72]. With support for thread and synchronization hierarchies, Paradyn-J can provide performance data in terms of individual AP threads.

Support for multi-threaded Java would be a good first step towards supporting the measure-

ment of parallel and distributed Java applications; we would already be able to measure multi-threaded Java applications run on SMPs. However, there are also several parallel message-passing implementations for Java based on PVM or MPI [67, 19, 11], there are meta-computing environments that use Java [4, 8, 20], and there is Sun's Remote Methods Interface [64] for client/server Java applications. To port Paradyn-J to one or more of these systems, we need to add support to Paradyn-J to discover the participating processes in these applications, and to extract synchronization information from the classes and/or libraries that implement the communication between the AP's processes. As a result, we could apply our performance measurement techniques to more complicated Java executions including parallel and distributed applications that are run across the Web. In these types of environments there are some interesting questions related to what type of performance data is useful to a program developer, and related to how to measure and collect performance data for applications consisting of potentially thousands of processes distributed across nodes on the Web.

We would also like to port Paradyn-J to a real dynamic compiler. Currently this is not possible because we need VM source code to port Paradyn-J, and no Java dynamic compiler source code is available to us. However, Sun plans to make its HotSpot source code available for research use, which would allow us to test our resource mapping functions and multiple execution form representations in the presence of a real optimizing run-time compiler.

Another area for future investigation is to examine in more detail the VM-developer's view of the execution. In particular, examining the use of performance data to tune a dynamic compiler's run-time compiling heuristic. For example, performance data associated with the VM's execution of particular application byte-codes may yield a definable set of AP code characteristics that either do or do not perform well from run-time compilation. If these characteristics can be incorporated into the run-time compiler's heuristics, then potentially better decisions can be made about which methods to compile, and when to compile them at run-time.

Finally, there are some interesting questions of how performance data collected at run-time could be used to drive the dynamic compiler's run-time compiling heuristics and be used to produce better optimized native code. If performance data collection becomes part of the normal execution of a dynamic compiler, then profiling costs must be recovered by the execution time saved due to using the data to make better compiling decisions. An analysis of collecting and using different types of performance data in the run-time compiler may yield important results in the area of run-time compiler optimizations and heuristics for triggering run-time compilation.

References

[1] Vikram S. Adve, Jhy-Chun Wang, John Mellor-Crummey, Daniel A. Reed, Mark Anderson, and Ken Kennedy. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Proceedings of the Supercomputing'95 Conference*, San Diego, California, December 1995.

[2] AT&T. *System V Interface Definition, Third Edition*. Addison-Wesley, 1989.

[3] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, Effective Dynamic Compilation. In *Proceedings of ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, Pennsylvania, May 1996.

[4] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the Web. In *Proceedings of the Ninth International Conference on Parallel and Distributed Computing (PDCS)*, September 1996.

[5] Robert Bedichek. Some Efficient Architecture Simulation Techniques. In *Proceedings of the Winter '90 USENIX Conference*, pages 53–63, Washington, D. C., January 1990.

[6] Bob Boothe. Fast Accurate Simulation of Large Shared Memory Multiprocessors (revised version). Technical Report UCB/CSD-93-752, Computer Science Division, University of California, Berkeley, June 1993.

[7] Gary Brooks, Gilbert J. Hansen, and Steve Simmons. A New Approach to Debugging Optimized Code. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 1–11, San Francisco, California, June 1992.

[8] Sean P. Brydon, Paul Kmiec, Michael O. Nearly, Sami Rollins, and Peter Cappello. Javelin++ Scalability Issues in Global Computing. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 171–180, San Francisco, California, June 1999.

[9] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeno Dynamic Optimizing Compiler for Java. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, California, June 1999.

[10] Bryan M. Cantrill and Thomas W. Doeppner Jr. ThreadMon: A Tool for Monitoring Multithreaded Program Performance. http://www.cs.brown.edu/research/thmon/thmon.html.

[11] Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. Object Serialization for Marshalling Data in an Interface to MPI. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 66–71, San Francisco, California, June 1999.

[12] Bryan Carpenter, Guansong Zhang, Geoffrey Fox, Xiaoming Li, Xinying Li, and Yuhong Wen. Towards a Java Environment for SPMD Programming. In *Proceedings of the 4th International Europar Conference*, pages 659–668, Southampton, UK, September 1998.

[13] Bob Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems, Nashville, Tennessee*, pages 128–137, May 1994.

[14] Compaq Computer Corporation. Compaq DIGITAL FX!32. White paper: http://www.digital.com/amt/fx32/fx-white.html.

[15] Brian F. Cooper, Han B. Lee, and Benjamin G. Zorn. ProfBuilder: A Package for Rapidly Building Java Execution Profilers. Technical report, University of Colorado, April 1998.

[16] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling Java Just In Time. *IEEE MICRO, Vol. 17 No. 3*, pages 36–43, May/June 1997.

[17] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor Simulation and Tracing using Tango. In *Proceedings of the International Conference on Parallel Processing*, pages 99–107, August 1991.

[18] L. Peter Deutsch and Alan Shiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages (POPL)*, pages 297–302, Salt Lake City, Utah, January 1984.

[19] Adam J. Ferrari. JPVM: Network Parallel Computing in Java. ACM 1998 Workshop on Java for High-Performance Network Computing, Poster Session, Palo Alto, California, February 1998.

[20] Ian Foster and Steven Tuecke. Enabling Technologies for Web-Based Ubiquitous Supercomputing. In *Proceedings of the 5th IEEE Symposium on High Performance Distributed Computing*, pages 112–119, Syracuse, New York, August 1996.

[21] J. Steven Fritzinger and Marianne Mueller. Java Security. Sun Microsystems Inc. White Paper, 1996.

[22] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a Call Graph Execution Profiler. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, June 1982.

[23] David Griswold. The Java HotSpot Virtual Machine Architecture. Sun Microsystems Whitepaper, March 1998.

[24] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leeks and Access Errors. In *Proceedings of the Winter'92 USENIX Technical Conference*, pages 125–136, San Francisco, California, January 1992.

[25] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 4(3):29–39, September 1991.

[26] John L. Hennessy. Symbolic Debugging of Optimized Code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):323–344, July 1982.

[27] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic Program Instrumentation for Scalable Performance Tools. In *Proceedings of the Scalable High-performance Computing Conference (SHPCC)*, Knoxville, Tennessee, May 1994.

[28] Jeffrey K. Hollingsworth, Barton P. Miller, Marcelo J. R. Goncalves, Oscar Naim, Zhichen Xu, and Ling Zheng. MDL: A Language and Compiler for Dynamic Program Instrumentation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, San Francisco, California, November 1997.

[29] Urs Holzle and David Ungar. A Third-Generation Self Implementation: Reconciling Responsiveness with Performance. In *Proceedings of the 9th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 229–243, Portland, OR, October 1994.

[30] IBM Corporation. Jinsight. http://www.alphaWorks.ibm.com/formula/jinsight, April 1998.

[31] Intel Corporation. VTune. http://www.intel.com/vtune/analyzer/, March 1998.

[32] Intuitive Systems Inc. OptimizeIt. Sunnyvale, California. http://www.optimizeit.com/, March 1999.

[33] R. Bruce Irvin and Barton P. Miller. Mechanisms for Mapping High-Level Parallel Performance Data. In *Proceedings of the ICPP Workshop on Challenges for Parallel Processing*, Chicago, Illinois, August 1996.

[34] Guy Lewis Steele Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. MIT Artificial Intelligence Memo 452, January 1978.

[35] Mark Kantrowitz. Portable Utilities for Common Lisp, User Guide and Implementation Notes. Technical Report CMU-CS-91-143, Carnegie Mellon University, May 1991.

[36] John G. Kemeny and Thomas E. Kurtz. *Basic Programming*. John Wiley & Sons, Inc. Publisher, 1967.

[37] KL Group. JProbe. Toronto, Ontario Canada. http://www.klg.com/jprobe/.

[38] Robert A. Kowalski. Predicate Logic as Programming Language. *IFIP Congress*, pages 569–574, 1974.

[39] Geoffrey H. Kuenning. Precise Interactive Measurement of Operating Systems Kernels, Software. *Software Practice and Experience. D. E. Comer and A. J. Wellings editors. Wiley publisher.*, 25(1):1–21, January 1995.

[40] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[41] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley, 1996.

[42] Lucent Technologies, Inc. Inferno Release 2.0 Reference Manual. http://www.lucent-inferno.com/Pages/Developers/Documentation/Ref_Man20/index.html, 1997.

[43] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, and F. Bodin. Performance Analysis of pC++: A Portable Data-Parallel Programming System for Scalable Parallel Computers. In *Proceedings of the 8th International Parallel Processing Symposium (IPPS)*, pages 75–85, Cancun, Mexico, April 1994.

[44] Carmine Mangione. Performance test show Java as fast as C++. *JavaWorld http://www.javaworld.com/*, February 1998.

[45] Margaret Martonosi, Anoop Gupta, and Thomas E. Anderson. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *Proceedings of the 1992 ACM SIGMETRICS and Performance '92 International Conference on Measurement and Modeling of Computer Systems*, pages 1–12, June 1992.

[46] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, April 1960.

[47] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer 28, 11*, November 1995.

[48] John K. Ousterhout. Tcl: An Embeddable Command Language. In *Proceedings of the Winter'90 USENIX Conference*, pages 133–146, Washington, D. C., January 1990.

[49] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *3rd Workshop on Computer Architecture Education*, February 1997.

[50] Srinivansan Parthasarathy, Michal Cierniak, and Wei Li. NetProf: Network-based High-level Profiling of Java Bytecode. Technical Report 622, University of Rochester, May 1996.

[51] Sharon E. Perl, William E. Weihl, and Brian Noble. Continuous Monitoring and Performance Specification. *Compaq Systems Research Center (SRC) Research Report*, June 26 1998.

[52] Todd A. Proebsting. Optimizing an ANSI C Interpreter with Superoperators. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 322–332, San Francisco, California, January 1995.

[53] Rational Software Corporation. Visual Quantify. Cupertino, California.

[54] Rational Software Corporation. Quantify User's Guide. Cupertino, California, 1993.

[55] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley W. Schwartz, and Luis F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.

[56] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, pages 48–60, Santa Clara, California, June 1993.

[57] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, 1995.

[58] Mark Roulo. Accelerate your Java apps! *JavaWorld http://www.javaworld.com/*, September 1998.

[59] Walter J. Savitch. *PASCAL An Introduction to the Art and Science of Programming*. The Benjamin/Cummings Publishing Co., Inc., 1984.

[60] Sun Microsystems Inc. Java built-in profiling system. On-line Java Tools Reference Page, http://www.javasoft.com/products/JDK/tools.

[61] Sun Microsystems Inc. Java Native Interface (JNI). http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html.

[62] Sun Microsystems Inc. Java Virtual Machine Profiler Interface (JVMPI). http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html.

[63] Sun Microsystems Inc. The Java 2 Platform (1.2 release of the JDK). http://java.sun.com/jdk/, 1999.

[64] Sun Microsystems Inc. Java Remote Method Invocation-Distributed Computing for Java. White Paper, May 1999.

[65] Ariel Tamches and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation Conference (OSDI)*, New Orleans, February 1999.

[66] Thinking Machines Corporation, Cambridge Massachusetts. CM Fortran Reference Manual. January 1991.

[67] David Thurman. jpvm. http://www.isye.gatech.edu/chmsr/jPVM/.

[68] Guido van Rossum. Python Reference Manual. Corporation for National Research Initiatives (CNRI), Reston, VA., 1999.

[69] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl, Second Edition*. O'Reilly & Associates, Inc., 1996.

[70] Winifred Williams, Timothy Hoel, and Douglas Pase. The MPP Apprentice Performance Tool. *Programming Environments for Massively Parallel Distributed Systems. Karsten M. Decker and Rene M. Rehmann, editors. Birkhauser Verlag, Publisher*, 1994.

[71] Zhichen Xu, James R. Larus, and Barton P. Miller. Shared-Memory Performance Profiling. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Las Vegas, Nevada, June 1996.

[72] Zhichen Xu, Barton P. Miller, and Oscar Naim. Dynamic Instrumentation of Threaded Applications. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 49–59, Atlanta, Georgia, May 1999.

[73] J. C. Yan. Performance Tuning with AIMS – An Automated Instrumentation and Monitoring System for Multicomputers. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, pages 625–633, Wailea, Hawaii, January 1994.

[74] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System Support for Automatic Profiling and Optimization. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, Saint-Malo, France, October 1997.