Performance Measurement of Dynamically Compiled Java Executions

Tia Newhall[1] and Barton P. Miller[2]

[1] Computer Sciences Program, Swarthmore College, Swarthmore, PA 19081 USA
[2] Computer Sciences Department, University of Wisconsin, Madison, WI 53706 USA

Abstract. With the development of dynamic compilers for Java, Java's performance promises to rival that of equivalent C/C++ binary executions. This should ensure that Java will become the platform of choice for ubiquitous Web-based supercomputing. Therefore, being able to build performance tools for dynamically compiled Java executions will become increasingly important. In this paper we discuss those aspects of dynamically compiled Java executions that make performance measurement difficult: (1) some Java application methods may be transformed from byte-code to native code at run-time; and (2), even in native form, application code may interact with the Java virtual machine. We describe Paradyn-J, an experimental version of the Paradyn Parallel Performance Tool that addresses this environment by describing performance data from dynamically compiled executions in terms of the multiple execution forms (interpreted byte-code and directly executed native code) of a method, costs of the dynamic compilation, and costs of residual dependencies of the application on the virtual machine. We use performance data from Paradyn-J to tune a Java application method, and improve its interpreted byte-code execution by 11% and its native form execution by 10%. As a result of tuning just one method, we improve the application's total execution time by 11% when run under Sun's ExactVM (included in the Platform2 release of JDK). The results of our work are a guide to virtual machine designers as to what type of performance data should be available through Java VM performance tool APIs.

1 Introduction

The platform independence of Java makes it ideal for ubiquitous web-based supercomputing. In most cases interpreted Java does not perform as well as equivalent native code [11]. For Java to compete, it is clear that it must execute, at least in part, in native form. Dynamic compilation is the most promising alternative for transforming Java byte-codes to native code. Thus, as more performance critical Java programs are developed and run by VMs that implement dynamic compilers, the ability to build performance tools for these types of executions will become increasingly important. We describe Paradyn-J, an experimental version of the Paradyn Parallel Performance Tool [12] that addresses this environment by dealing with the multiple execution forms (interpreted byte-code and directly executed native code) of a method, costs of the dynamic compilation, and costs of residual dependencies of the Java application program (AP) on the virtual machine (VM). Paradyn-J measures a simulated dynamically compiled execution of Java programs run under Sun's version 1.1.6 of the Java VM running on a SPARC-Solaris 2.6 platform. Paradyn-J generates and inserts byte-code and native code instrumentation into the VM and AP at run-time; it requires no changes to the VM binary nor to AP .class files prior to execution.

Figure 1 shows the two execution modes of an environment that uses dynamic compilation: (1) the VM interprets AP byte-codes; (2) native code versions of AP methods, that the VM compiles at run-time, 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 becomes more like a run-time library to the native form of an AP method. At any point in the execution, the VM may compile a method, and some methods may never be compiled; a dynamically compiled method can be interpreted in byte-code form, compiled and directly executed in native code form or both during its execution.

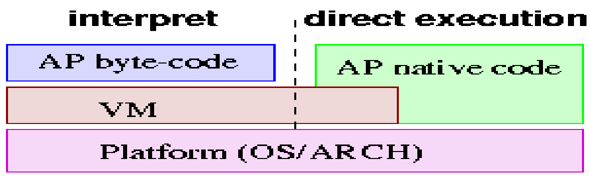


Fig. 1. During a dynamically compiled execution, methods may be interpreted by the VM and/or compiled into native code and directly executed. *The native code may still interact with the VM. In this case, the VM acts like a run-time library to the AP.*

There are several challenges associated with the unique characteristics of dynamically compiled executions that make performance measurement difficult:

- **Multiple execution forms of the Java application program:** Parts of the application program are transformed from byte-code to native code by the VM at run-time; as a result, the location and structure of Java application method code can change at run-time. From a performance measurement standpoint this causes two problems. First, a performance tool must be able to 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.
- **Run-time Transformations:** Compilation of Java byte-code to native code occurs at run-time. A performance tool must represent performance data associated with the transformational activities.
- **Interaction between the VM and the AP:** Even the native code methods interact with the VM (the VM acts more like a run-time library). Performance data that explicitly describe these VM-AP interactions will help a programmer better understand an application's execution.

We explicitly represent VM-AP interactions during the interpretation and direct execution of the AP, costs associated with the run-time compilation of Java byte-codes to native code, and the relationships between the different forms of AP code objects so that performance data from different forms of an AP method can be correlated.

To quantify the unique run-time costs associated with dynamic execution, we compare an all-interpreted execution to a dynamically compiled execution using Sun's ExactVM dynamic compiler included in the Platform 2 release of the JDK [19]. Our study (presented in Section 2) examines three cases where we suspect that a method's dynamic compilation will result in little or no improvement over its all-interpreted execution: (1) methods whose native form has a lot of interaction with the VM, (2) methods whose interpreted execution time is not dominated by interpreting byte-code, for example, methods dominated by I/O costs, (3) small methods with few byte-code instructions. Results from our study demonstrate the need for detailed performance data from dynamically compiled executions; we show how performance data that allows a user to compare the interpreted execution costs to the native execution costs, to see the VM costs associated with the native code's execution, and to see ratios of I/O time to interpreted and native execution times, can be used to more easily determine how to tune the AP to improve its performance. We discuss, in Section 5, how this same type of performance data could be used by a VM developer to tune the VM.

In Section 3, we describe a tool for measuring dynamically compiled Java executions. Paradyn-J provides the types of detailed performance data that we discovered were critical to understanding the performance of a dynamically compiled execution. In Section 4, we present results from using

Paradyn-J to measure a simulated dynamic execution of two Java applications. We show how Paradyn-J can profile VM overheads, I/O costs, interpretation costs, direct execution costs, and run-time compilation costs associated with the byte-code and the native code forms of individual methods in the Java application. We use performance data from Paradyn-J to tune a dynamically compiled Java application method, and improve its interpreted byte-code execution by 11% and its native form execution by 10%. Results of these studies point to places were JDK's new profiling interface, JVMPI [18], should be expanded to provide performance data that measure the multiple execution forms of AP code, the run-time transformation costs associated with dynamically compiling AP byte-codes, and the VM overhead associated with the VM's execution of AP native and byte-code.

2 Evaluating Dynamic compilation performance

Performance measurement of dynamically compiled executions is more complicated than that of statically compiled program executions; beyond the general need for detailed performance data, a performance tool needs to deal with the multiple execution forms of the AP, and with run-time interactions between the AP and the VM. In this section, we motivate the need for performance data that describe VM costs and other detailed run-time measures associated with interpreted byte-code and directly executed native code forms of a method. We show examples of how an AP developer can use such performance data to tune the AP. We demonstrate the need for these type of performance data by comparing total execution times of dynamically compiled and all-interpreted executions of three Java applications. We examine three cases where 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 ($(n \times Interp) \leq (a \times Interp + Compile + b \times NativeEx)$). 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 ExactVM. We 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 it waits to compile a method until it has been called 15 times. As a result, the main loop method is immediately compiled (since it contains a loop), 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 create objects, delete objects (resulting in increased garbage collection), or modify objects (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 allocation of 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 was 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 1. For about the first 3,000 iterations, interpreted execution performs better than dynamically compiled execution. After this, the costs of run-time 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 on Page 3, the $b \times NativeEx$ term can be written as $b \times (DirectEx + VMInteraction)$. In this application, it is likely that the $VMInteraction$ term dominates this expression, and as a result, dynamic compilation does not result in much performance improvement. 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. Our hypothesis was 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 (number of main loop iterations) are shown as Case 2 in Table 1. After about 500 iterations, the dynamically compiled execution performs better than the all-interpreted execution. Speed ups obtained for an increasing number of iterations are not that great; I/O costs dominate the native code's execution time[2]. The decrease in speed up values between the 10,000 and 100,000 iteration case is due to 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 these methods, the time

[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 speed up of 4.9 vs. a speed up of 1.04 for Case 1)

[2] 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)

spent interpreting method byte-codes is small, so the execution of a 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 was that dynamic compilation of these three small methods would not result in much improvement because their interpreted execution is not that expensive.

The results (Case 3 in Table 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 of the VM to interpret method call instructions vs. the cost of directly executing a native code call instruction[3]. 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.

Case 1: Object Modifications				**Case 2: I/O Intensive**				**Case 3: Small Methods**			
Iterations	Dyn Comp	Interp	Speed up	Iterations	Dyn Comp	Interp	Speed up	Iterations	Dyn Comp	Interp	Speed up
100,000	114.70	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

Table 1. Execution time (in seconds) of each Java kernel run by ExactVM comparing interpreted Java (Interp column) to dynamically compiled Java (Dyn Comp column). The results are the average of 10 runs, with standard deviations near 1%.

The result of this study points to specific examples where detailed performance measures from a dynamically compiled execution can provide information that is critical to understanding the execution. For real Java applications consisting of thousands of methods, some with complicated control flow structure, a performance tool that can represent specific VM 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. In Section 5, we discuss the implications of this study for a VM developer.

3 A Performance Tool for dynamically Compiled Java

We present Paradyn-J, a prototype implementation of a performance tool for measuring dynamically compiled Java executions. Paradyn-J generates and inserts (or removes) instrumentation code into AP and VM code at run-time; as a result, Paradyn-J requires no modifications to the VM nor to the AP prior to execution. We wanted to implement Paradyn-J to measure a real Java dynamic compiler, unfortunately, no source code was available for ExactVM [19] or HotSpot [4]. Instead, we simulated dynamic compilation, and built Paradyn-J to measure its execution. We first present our simulation and then the details of Paradyn-J's implementation.

[3] 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 overhead to handle method calls and returns). For 10,000,000 iterations there was a speed up of 31.8, and for a version with no method call overhead (all the code is in the main loop) a speed up of 11.2.

3.1 Simulation of a Dynamic Compiler

Our simulation approximates the three main run-time activities in a dynamically compiled execution: interpretation of method byte-code; run-time compilation of some methods; and direct execution of the native form of transformed methods. We simulate dynamic compilation by modifying the Java application and running it with a Java 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" method is replaced with a wrapper function that initially calls a byte-code version of the method. After we reach a threshold (based on number of calls) the wrapper calls a routine that simulates the method's run-time compilation. The "compiling" routine takes an estimated compiling time as a parameter, and it 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 [17]. It is compiled into a shared object that the VM loads at run-time. We approximated each method's compile time by timing ExactVM's run-time compilation of each method.

3.2 Dynamic Instrumentation for VM Code

Paradyn-J uses Paradyn's dynamic instrumentation [5] 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 using their own register window. This way instrumentation code will not destroy the values in the function's stack frame or registers. Figure 2 shows an example of dynamically instrumenting a VM function.

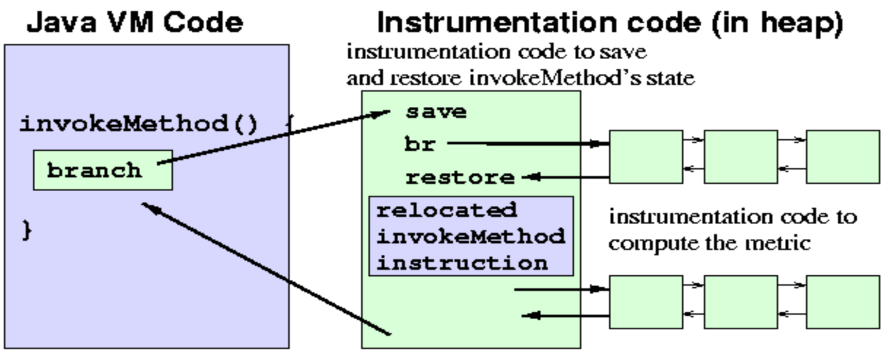


Fig. 2. Dynamic Instrumentation for Java VM code. *In this example VM function* `invokeMethod` *is instrumented. An instruction in* `invokeMethod` *is replaced with a branch instruction that jumps to the instrumentation code in the heap, and the overwritten* `invokeMethod` *instruction is relocated to the instrumentation code.*

3.3 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.

One approach to safely executing instrumentation code is to use a method call instruction to jump to byte-code instrumentation. 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, interpreting 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 this approach for safely executing instrumentation byte-codes. 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 3): the first time an instrumentation request is made for a method, the method is relocated to the heap and its size is expanded by adding `nop` byte-code instructions around each instrumentation point (currently, the instrumentation points are method entry, call sites, and method return points). 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 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.

The VM will automatically create a new execution stack frame and operand stack for our 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 constantpool of the calling class, the constantpool of the class must be modified to add entries that provide information about `do_baseTramp`. The constantpool of a class only has to be modified once (when the class file is loaded by the VM), 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; it is not safe to overwrite an instrumentation point with instrumentation code if the VM is currently executing instructions at the instrumentation point. 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 delay the insertion of byte-code 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.

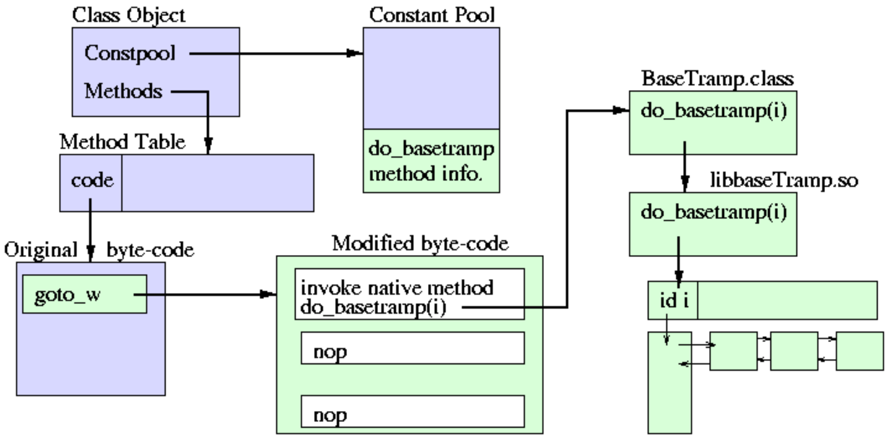


Fig. 3. Transformational Instrumentation for Java application byte-codes. *The darker colored boxes represent pre-instrumented Java VM data structures, the lighter colored boxes are added to instrument a Java Method.*

We re-use much of Paradyn's code generation facilities for generating SPARC 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. An 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. However, in our current implementation, we use an approach similar to the second option. 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. For example, to measure the amount of object creation overhead associated with objects created in AP method `foo`, some SPARC instrumentation code has to be called from instrumentation points in AP method `foo`, and some SPARC instrumentation code has to be inserted into VM code that performs object creation. The tag types may also be required to generate different instrumentation code. For example, return values for SPARC routines are stored in a register, and 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.

3.4 Paradyn-J's Interaction with the Java VM

Paradyn-J interacts with the Java VM at run-time whenever the VM loads a Java AP class file or compiles an AP method. New class files can be loaded by the Java VM at any point in the 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 AP 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 in VM data structures.

Once the VM has loaded a Java class, Paradyn-J examines the VM's data structures to discover the newly loaded methods for the class, and parses each method's byte-code instructions to 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.

Paradyn-J also interacts with the VM's run-time compiling routines. Paradyn-J discovers the native form of a compiled method so that the native form can be instrumented, creates mappings between byte-code and native code forms of a method so that performance data collected in different forms of an AP method can be correlated, and measures costs associated with the run-time compilation of a method. Paradyn-J instruments our routine that simulates run-time compilation. The instrumentation notifies Paradyn-J when a method is "dynamically compiled". At run-time, the VM calls `dlopen` to load the shared objects that contain the native versions of the AP methods and 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. Instrumentation at its entry point starts a timer to measure the run-time compiling overhead. Instrumentation at its exit point stops the timer measuring the compiling overhead, and gets the name of the native form of the method to obtain mapping information between the method's two forms.

For performance tools like ours that instrument AP byte-codes, there is a problem of how to deal with instrumented byte-codes that are about to be transformed by the VM's run-time compiler. One option is to let the VM compile the byte-code instrumentation along with the byte-code instructions of the AP method. 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. This requires that the performance tool interact with the VM immediately before and after compilation of a method. Since our simulated compiling routine does not actually translate byte-code to native code we did not have to worry about this problem for Paradyn-J's current implementation. However, when we port Paradyn-J to a real dynamic compiler we will have to handle this case.

4 Results

We present results using performance data from Paradyn-J. We demonstrate how Paradyn-J can 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 this data, 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.

Performance measures that describe specific VM-AP interactions are obtained by dynamically inserting instrumentation code into VM routines and Java AP routines to measure the interaction. For example, to measure the object creation overhead associated with objects created in AP method `foo`, we insert instrumentation into method `foo` that will set a `foo_flag` whenever `foo` creates an object, and we insert timer instrumentation into VM routines that handle object creates. The timer code to measure `foo`'s object creation overhead will be executed only when the `foo_flag` is set (only when the object is created by method `foo` will we execute `foo`'s timer code in the VM's object creation routines).

For the neural network program, we picked good candidate methods to "dynamically compile" by using Paradyn-J to measure its all-interpreted execution and choosing the seven application methods that were accounting for most of the execution time. We wrote JNI native versions and wrapper functions for each of these methods. Each method's compiling time was estimated by timing how long ExactVM takes to compile the method. We first demonstrate that Paradyn-J can associate performance data with AP methods in their byte-code and native code forms, and with the run-time compilation of AP methods. Figure 4 shows a performance visualization from Paradyn-J. The visualization is a time plot showing the fraction of CPU time per second for the byte-code (in black) and native (in white) forms of the `updateWeights` AP method, showing that `updateWeights` benefits from dynamic compilation. Figure 5 is a table visualization that shows performance measures of total CPU time (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 run-time; if the method was not "dynamically compiled", and instead was interpreted for each of these 69 calls, then the total CPU time would be 2.2 seconds (69 calls $\times$ 0.031 seconds/call). The total CPU 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).

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, 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), it 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 run-time, and its native code form (bottom row) is called 50 times, creates 600 objects, and accumulates 20.8 seconds of CPU time[4]. Its native form execution is more expensive (at 416 ms per call) than its interpreted execution (at 264 ms per call). These performance data tell the Java application 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 its performance is to try to reduce the number of objects created in the method's 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 7 shows total CPU time and object

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

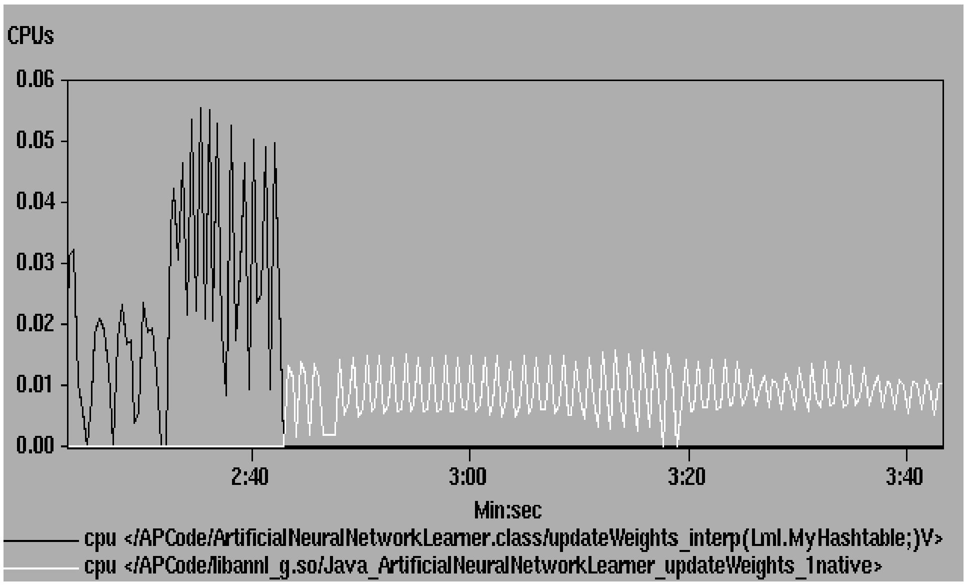
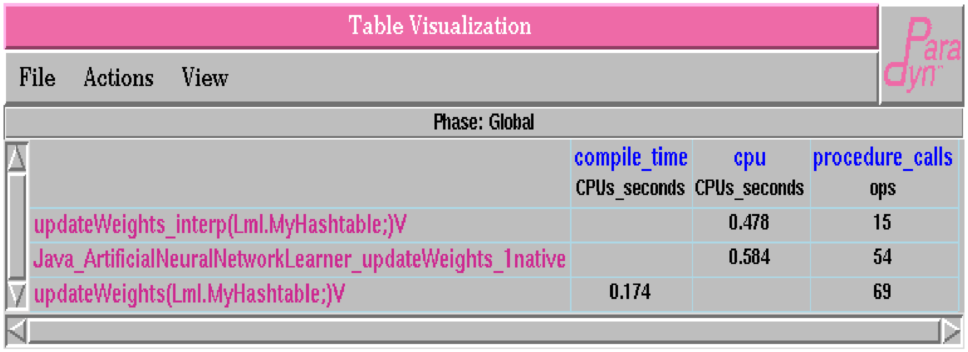


Fig. 4. Performance data for the `updateWeights` method from the dynamically compiled neural network Java application. *The time plot visualization shows the fraction of CPU time/second for the native (white) and byte-code (black) form of the method.*



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

Fig. 5. Performance data for the `updateWeights` method from the dynamically compiled neural network Java application. *The table shows the performance measures total CPU time (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).*

creates of the modified version of `calculateHiddenLayer`. The performance data shown in Figure 6 and Figure 7 were taken part way 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 CPU time executing the method's native code form improved by 10% (18.7 vs. 20.8 seconds).

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 execution time improved by 10% when run by ExactVM with dynamic compiling, and by 6% when run by ExactVM with dynamic

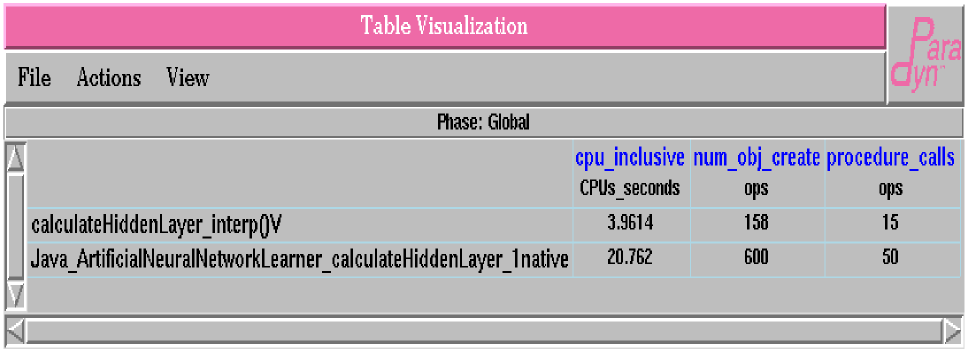


Table Visualization

File Actions View

Phase: Global

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

Fig. 6. 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.*

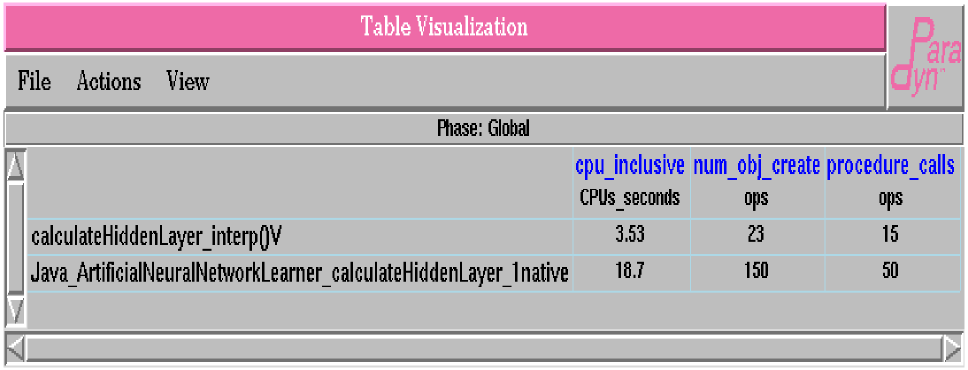


Table Visualization

File Actions View

Phase: Global

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

Fig. 7. Performance data for method `calculateHiddenLayer` after removing some object creates. *This table shows that the total CPU time for both the native and byte-code forms of the method is reduced as a result of reducing the number of object creates.*

compiling disabled (Table 2). 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"). One explanation for these results is that 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 overheads 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 "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 these methods would be available for performance tuning. Performance data from our tool that can measure VM overheads associated with the byte-code and native code form of a method help a program developer focus in on those methods to tune and give an indication of how to tune the method to improve its performance.

In general, for a method that does not benefit from being compiled at run-time by the VM, performance data that help explain why the method does not perform well will help a program developer more easily determine how to tune the method's performance. For example, in Section 2

	Original	Tuned	Change
Dynamic Compilation	21.09	18.97	11%
All-Interpreted	190.80	179.90	6%

Table 2. Total execution times (in seconds) under ExactVM for the original and the tuned versions of the neural network program. We improve the performance by 11% with dynamic compiling, and by 6% with dynamic compiling disabled (all-interpreted).

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 know how to tune the method to improve its performance.

In the 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 Section 2; we picked methods to "dynamically compile" based on the three cases we examined in Section 2. 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 did 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 3, we show performance data from Paradyn-J's measurement of each of the three methods.

Case 1: Object Modifications	
Measurement	Byte code
Total CPU seconds	2.352
Object Creation Overhead seconds	1.573

Case 2: I/O Intensive		
Measurement	Native	Byte-code
Total I/O seconds	5.649	0.3666
Total CPU seconds	0.005	0.0440

Case 3: Small Methods		
Measurement	Native code	Byte code
CPU time seconds/call	4.9 μs	6.7 μs
Method call seconds/call		2.5 μs

Table 3. Performance data from Paradyn-J's measurement of the CPU Simulation AP. These are detailed performance measures of methods in the AP that have performance characteristics similar to the three test cases from Section 2. Results for case 1: Object Modifications are in the left table, case 2: I/O Intensive are in the middle table, and case 3: Small Methods on the right.

For case 1, VM object creation overheads account for more than half of the method's execution time (1.573 out of 2.35 seconds, as shown in Table 3); 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 this 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)[5]. These performance data tell an AP developer to focus on reducing the I/O

[5] 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 represent these numbers as total values rather than per call values because each call to the method writes

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 represents CPU times for both the byte-code and native code form of the method. These data provide us with some explanation of why the 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 (an additional 2.5 μs for every 6.7 μs of interpreting byte-code) make interpreted execution much more expensive. 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. In a previous paper [13], we presented a performance tuning study of an all-interpreted execution of this Java application. In this study we reduce method call overheads by tuning the application to remove some method calls. Performance data from our tool led us to easily determine which methods to tune and which calls to remove from the execution to improve its performance. The performance data from these three methods describe the detailed behaviors needed by AP developers to tune their dynamically compiled applications.

Results of these studies demonstrate the need for performance data from dynamically compiled Java executions that measure the multiple execution forms of AP code, the run-time transformation costs associated with dynamically compiling AP byte-codes, and VM overhead associated with interactions with the execution of AP native and byte-code. This work can act as a concrete guide for VM developers as what to include in VM profiling interfaces (such as JVMPI) to provide Java application developers with information that is critical to understanding their application's performance when run by a dynamic compiler VM.

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 overheads are expensive can be used to tune the compiler to aggressively in-line methods (this is 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 overheads) 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 run-time, could be used to drive the VM's run-time 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 run-time information into compilers to produce better optimized versions of code and/or to drive run-time compiling heuristics [21], [6], [1], [2] (these are all for languages other than Java).

a different number of bytes; they are not directly comparable on a per call basis

6 Related Work

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. 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.

Some tools instrument the application byte-code (NetProf [14] and ProfBuilder [3]) prior to execution by the VM. NetProf and ProfBuilder re-write 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.

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[6], Java class file loading, garbage collection, and run-time compilation. Second, there is no way to obtain 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 [10], JDK's VM [16], Visual Quantify [15], and Jinsight [7]), or they interact with the Java VM at run-time using API's implemented by the VM (OptimizeIt [9], and VTune [8]).

An example of a profiling tool that is implemented as a special version of the Java VM is JProbe. JProbe, profiles interpreted and JIT compiled Java. It provides cumulative CPU times and counts associated with application methods, and counts associated with object creates. It also provides call graph and memory usage displays (showing memory allocation and garbage collection statistics as the application runs). An 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. OptimizeIt starts all Java applications that it measures, and uses low-level API's in the Java VM to obtain performance data for the application. For Sun's Java 2 Platform version of the VM, OptimizeIt uses the new Java profiling interface JVMPI [18] to obtain its data.

All of these profiling tools represent performance data in term of the interpreted or JIT compiled application's execution. Some provide measures of specific VM costs associated with the application's execution. For example, JProbe, OptimizeIt and Visual Quantify provide some memory profiling information in the form of garbage collection and object instance creation counts. However, in all of these tools most of the VM is hidden and as a result, these tools cannot describe performance data in terms of arbitrary interactions between the VM and the Java application. Also, there is no tool

[6] The timing instrumentation used by these tools is not thread aware.

that we know of for profiling dynamically compiled Java executions. Paradyn-J is the only one that can represent arbitrary VM-AP interactions, VM and other run-time costs associated with byte-code and native code forms of an AP method, and performance measures associated with the run-time compilation of AP methods.

7 Conclusions and Future work

In this paper, we discussed some of the unique characteristics of dynamically compiled Java executions that make performance measurement difficult. We described a prototype implementation of a performance tool for measuring dynamically compiled Java executions that addresses these problems by dealing with the multiple execution forms (interpreted byte-code and directly executed native code) of a method, costs of the dynamic compilation, and costs of residual dependencies of the Java application program on the virtual machine. We used Paradyn-J to demonstrate how we can represent data that is critical to understanding the performance of dynamically compiled executions; performance data from Paradyn-J can be used by a Java application developer or by a Java virtual machine developer to more easily determine how to tune the Java application or the Java virtual machine. Our work is a guide for what type of performance data should be available through Java VM performance tool interfaces.

For Paradyn-J to be more useful to developers of high performance Java applications, we need to add support for profiling threaded Java programs. In future versions of Paradyn-J, we will support threaded Java applications by leveraging off of Paradyn's new support for threads [20].

8 Acknowledgments

We thank Marvin Solomon and Andrew Prock for providing the Java application programs used in Section 4, and we thank Karen Karavanic, Brian Wylie, and Zhichen Xu for their comments on this manuscript. This work is supported in part by Department of Energy Grant DE-FG02-93ER25176, NSF grants CDA-9623632 and EIA-9870684, and DARPA contract N66001-97-C-8532. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. 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.
2. Compaq Computer Corporation. Compaq DIGITAL FX!32. White paper: http://www.digital.com/amt/fx32/fx-white.html.
3. 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.
4. David Griswold. The Java HotSpot Virtual Machine Architecture. Sun Microsystems Whitepaper, March 1998.
5. 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.
6. Urs Hölzle 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.
7. IBM Corporation. Jinsight. http://www.alphaWorks.ibm.com/formula/jinsight, April 1998.
8. Intel Corporation. VTune. http://www.intel.com/vtune/analyzer/, March 1998.
9. Intuitive Systems Inc. Optimize It. Sunnyvale, CA, http://www.optimizeit.com/.

10. KL Group. JProbe. Toronto, Ontario Canada. http://www.klg.com/jprobe/.
11. Carmine Mangione. Performance test show Java as fast as C++. *JavaWorld http://www.javaworld.com/*, February 1998.
12. 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.
13. Tia Newhall and Barton P. Miller. Performance Measurement of Interpreted Programs. *EuroPar'98, Southampton, UK*, pages 146–156, September 1998.
14. Srinivansan Parthasarathy, Michal Cierniak, and Wei Li. NetProf: Network-based High-level Profiling of Java Bytecode. Technical Report 622, University of Rochester, May 1996.
15. Rational Software Corporation. Visual Quantify. Cupertino, California.
16. Sun Microsystems Inc. Java built-in profiling system. On-line Java Tools Reference Page, http://www.javasoft.com/products/JDK/tools.
17. Sun Microsystems Inc. Java Native Interface (JNI). http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html.
18. Sun Microsystems Inc. Java Virtual Machine Profiler Interface (JVMPI). http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html.
19. Sun Microsystems Inc. The Java 2 Platform (1.2 release of the JDK). http://java.sun.com/jdk/, 1999.
20. 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.
21. 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.

This article was processed using the LaTeX macro package with LLNCS style