Performance Measurement of Dynamically Compiled Java Executions

Tia Newhall and Barton P. Miller
University of Wisconsin Madison
Madison, WI 53706-1685 USA
+1 (608) 262-1204
{newhall,bart}@cs.wisc.edu

1 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 runtime; 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 10% when run under Sun's ExactVM (included in the Platform2 release of JDK).

1.1 Keywords

Performance profiling tool, dynamic compilation, Java.

2 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 [1]. 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 [2] 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 simulated dynamically compiled Java programs run under Sun's version 1.1.6 of the Java VM. Paradyn-J generates and inserts byte-code and native code instrumentation into the VM and AP at runtime; 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 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 becomes more like a runtime 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.

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

1. 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 runtime; as a result, the location and structure of Java application method code can change at runtime. 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.

2. Runtime Transformations: Compilation of Java byte-code to native code occurs at runtime. A performance tool must represent performance data associated with the transformational activities.

3. Interaction between the VM and the AP: Even the native code methods interact with the VM (the VM acts more like a runtime library). Performance data that explicitly describes these VM-AP interactions will help a programer better understand their application's execution.

We explicitly represent VM-AP interactions during the interpretation and direct execution of the AP, costs associated with the runtime 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.

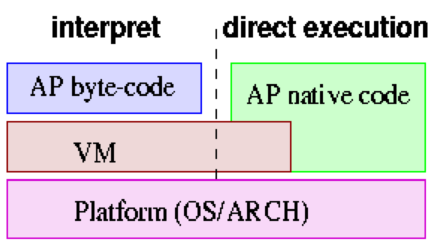


Figure 1: During a dynamically compiled execution methods may be interpreted by the VM and/or compiled into native code and directly executed. ***The native code may still interact with the VM. In this case, the VM acts like a runtime library to the AP.***

To quantify the unique runtime 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 JDK (java.sun.com/jdk/). Our study (presented in Section 3) 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) small methods with few byte-code instructions, (2) methods whose native form has a lot of interaction with the VM, (3) methods whose interpreted execution time is not dominated by interpreting byte-code, for example, methods dominated by I/O costs. 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 6, how this same type of performance data could be used by a VM developer to tune the VM.

In Section 4, 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 5, 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 runtime 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%. InSection 8, we discuss implementation issues associated with adding support to Paradyn-J for measuring parallel Java.

3 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 runtime interactions between the AP and the VM. In this section, we motivate the need for performance data that describe VM costs and other detailed runtime 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 this 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
$((n \times Interp) < (a \times Interp + Compile + b \times NativeEx))$.
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 do 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 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 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. 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. 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 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 will 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. Part of this is due to the high costs of VM overhead to interpret method call instructions vs. the cost of directly executing a native code call instruction. Performance data that explicitly represent VM method call overheads, VM costs to interpret

Case 1: object modifications			
iterations	Dyn Comp	Interp	Speed up
100,000	114.7	119.5	1.04
10,000	1.73	2.04	1.18
1,000	0.71	0.65	0.91
100	0.70	0.63	0.90

Case 2: I/O intensive			
iterations	Dyn Comp	Interp	Speed up
100,000	427.1	436.43	1.02
10,000	40.47	42.70	1.05
1,000	4.53	4.64	1.02
100	1.06	0.99	0.94

Case 3: small methods			
iterations	Dyn Comp	Interp	Speed up
10,000,000	1.76	35.11	19.94
1,000,000	0.83	4.16	5.01
100,000	0.74	0.98	1.32
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). *Each measurement is the average of 10 runs.*

byte-codes, and VM costs to execute native code could be used by an AP developer to identify that interpreted call instructions are expensive.

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 6, we discuss the implications of this study for a VM developer.

4 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 runtime; 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 or HotSpot[3]. Instead, we simulated dynamic compilation, and built our prototype to measure its execution. We first present our simulation and then the details of Paradyn-J's implementation.

4.1 Simulation of a Dynamic Compiler

Our simulation approximates the three main runtime 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 runtime 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 (java.sun.com/products/jdk/1.2/docs/guide/jni/index.html). It is compiled into a shared object that the VM loads at runtime. We approximated each method's compile time by timing ExactVM's runtime compilation of each method.

We simulate the three different execution phases we need to demonstrate a performance tool that can provide detailed performance measures from a dynamically compiled execution. However, we do not use our simulation to compare total execution times of a dynamically compiled execution to an interpreted execution because each wrapper function adds an extra layer of indirection for calls to byte-code and JNI native code versions of the method. The wrapper functions also add more interpreted execution since they are written in Java and interpreted each time they are called.

4.2 Performance Tool Implementation

Paradyn-J is an extension of our earlier tool for measuring interpreted Java executions [4]. We modified this tool to add support for measuring our simulated dynamically compiled executions run under Sun's version 1.1.6 of JDK. Paradyn-J generates and inserts instrumentation code into the AP and VM at runtime; we use Paradyn's Dynamic Instrumentation [2] to instrument VM code and AP native code, and we use Transformational Instrumentation [4] to instrument AP byte-codes.

Paradyn-J interacts with the VM's runtime compiling routines. We discover the native form of a compiled method so that it can be instrumented, create 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 measure costs associated with the runtime compilation of a method. We instrument our routine that simulates runtime compilation. The instrumentation notifies the tool whenever a method is "dynamically compiled". At runtime, the VM calls dlopen() to load the shared objects that contain the native versions of the AP methods and contain our "compiling" routine. We instrument the VM to catch dlopen() events. When we detect that the VM has loaded our "compiling" routine, we instrument it. Instrumentation at its entry point starts a timer to measure the dynamic compiling time. Instrumentation at its exit point stops the timer measuring the compiling time, 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 dynamic 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 our tool's current implementation. However, when we port our tool to a real dynamic compiler we will have to handle this case.

5 RESULTS

We present results using performance data from Paradyn-J. We demonstrate how we 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.

We obtain performance measures that describe specific VM-AP interactions 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 will be executed only when the `foo_flag` is set (only when the object is created by method `foo` will we measure the VM overhead associated with the object create).

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. 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 2 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 black) and native (in white) forms of the `updateWeights` AP method, showing that `updateWeights` benefits from dynamic compilation. Figure 3 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). This 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 all of these 69 calls, then the total execution time would be 2.2 seconds (69 calls $\times$ 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).

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 4, the visualization shows a method (`calculateHiddenLayer`) that accounts for most of the object creates. This 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 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). This performance data tells the Java application

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

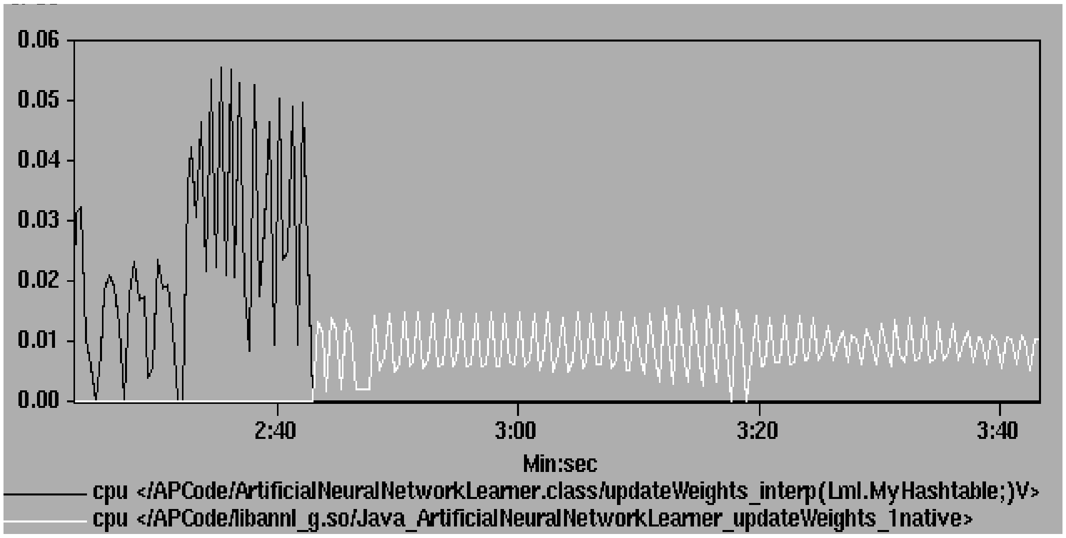
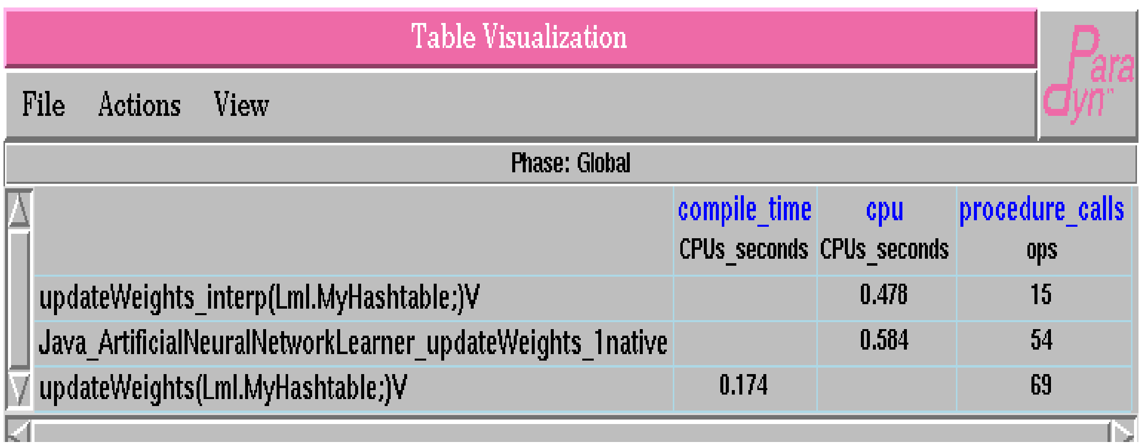


Figure 2: Performance data for the `updateWeights` method from the dynamically compiled neural network Java application. ***The time plot visualization shows the fraction of CPUtime/second for the native (white) and byte-code (black) form of the method.***



	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 3: 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).***

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 5 shows total CPUtime and object creates of the modified version of `calculateHiddenLayer.` This 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).

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 compiling

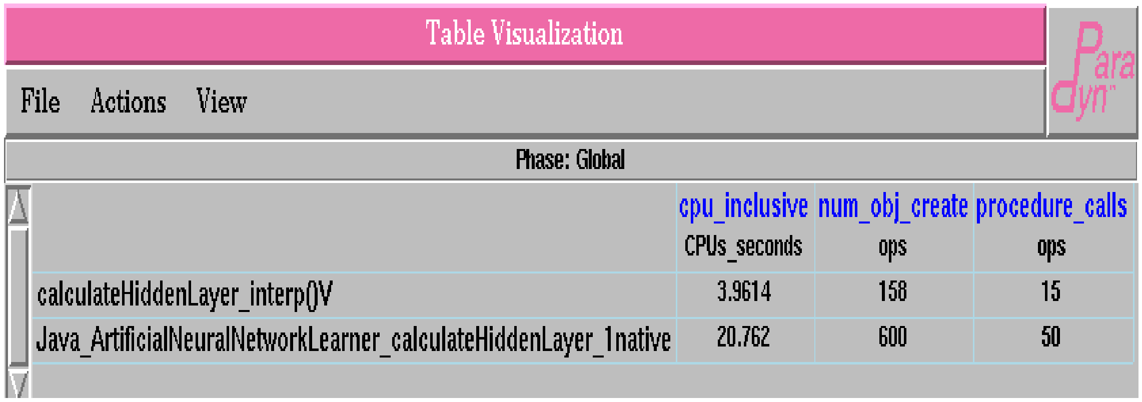


Figure 4: Performance data for method `calculateHiddenLayer`. ***The total CPU time (first column), total number of object creates (second column), and total number of calls (third column) to the byte-code (top row) and native code (bottom row) forms of the method.***

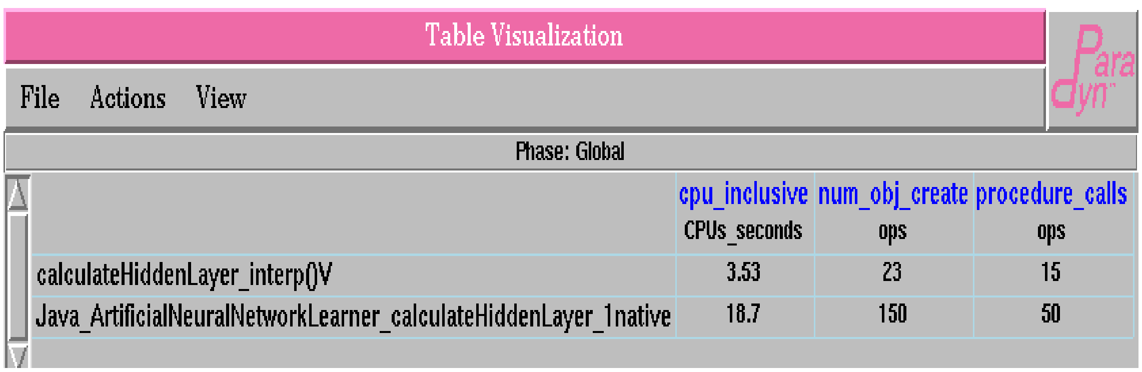


Figure 5: Performance data for method `calculateHiddenLayer` **after removing some object creates.** ***This table shows that the total CPUtime for both the native and byte-code forms of the method is reduced as a result of reducing the number of object creates.***

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"). 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 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%

Table 2: 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).***

In general, for methods that do 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 3 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.

Case 1: object creates		Case 2: I/O intensive			Case 3: small functions		
Measurement	**Byte-code**	**Measurement**	**Native**	**Byte-code**	**Measurement**	**Native**	**Byte-code**
Total CPU seconds	2.3515	***Total I/O seconds***	5.6493	0.36658	***CPU seconds***	4.9 µs	6.7 µs
Object Creation Overhead seconds	1.5730	***Total CPU seconds***	0.00496	0.04403	***MethodCall Time***		2.5 µs

Table 3: Performance data from 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 3.***

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 3; we picked methods to "dynamically compile" based on the three cases we examined in Section 3. 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 3 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.

For case 1, VM object creation overheads 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 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)[2]. This performance data tells an AP developer to focus on reducing the I/O costs since they account for the largest fraction of this 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. This data provides 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 (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 [4], 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.

6 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 runtime 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

[2] 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.

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 [5, 6, 7, 8] (these are all for languages other than Java).

7 RELATED WORK

There are several performance tools for interpreted or just-in-time (JIT) compiled Java. JDK's built in profiling system (www.javasoft.com/products/JDK/tools) provides total elapsed time and counts associated with Java application methods, and provides call graph information. JProbe (www.klg.com/jprobe/), is a special version of a VM for JDK versions up to 1.1.6. It can profile 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). Optimize It (www.optimizeit.com), is a Java profiler that works with Java applications interpreted by JDK's VM versions through 1.1.6. It provides total CPUtime measures associated with application threads, total CPUtime and counts associated with methods, a real-time memory profiler (number of instances per class), and a source code viewer annotated with method CPU times and counts.

These tools provide performance measures in terms of Java application code, and two of them provide some indication of how the application interacts with VM memory allocation and garbage collection. However, they can not represent performance data in terms of specific interactions between the VM and the Java application. These tools also do not currently support dynamically compiled Java executions. Paradyn-J is the only one that can represent arbitrary VM-AP interactions, VM and other runtime costs associated with byte-code and native code forms of an AP method, and performance measures associated with the runtime compilation of AP methods.

8 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.

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 [9].

9 ACKNOWLEDGEMENTS

We thank Marvin Solomon and Andrew Prock for providing the Java application programs used in Section 5, 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.

10 REFERENCES

[1] C. Mangione. Performance test show Java as fast as C++. *JavaWorld*, February 1998.

[2] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer 28, 11*, November 1995.

[3] D. Griswold. The Java HotSpot Virtual Machine Architecture. *Sun Microsystems Whitepaper,* March 1998.

[4] T. Newhall and B. P. Miller. Performance Measurement of Interpreted Programs. *EuroPar'98*, September 1998.

[5] J. B. Chen, M. D. Smith, and B. N. Bershad. Morph: A Framework for Platform-Specific Optimization. *White-Paper http://www.eecs.harvard.edu/morph/,* March 1996.

[6] U. Holzle and D. Ungar. A Third-Generation Self Implementation: Reconciling Responsiveness with Performance. In *Proceedings of the ACM OOPSLA '94 Conference, Portland, OR*, October 1994.

[7] Digital Semiconductor. FX!32 Technical Introduction. *On-line: http://www.digital.com/info/semiconductor/amt/fx32/*.

[8] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, Effective Dynamic Compilation. In *Proceedings of ACM PLDI Conference on Programming Language Design and Implementation*, May 1996.

[9] Z. Xu, B. P. Miller, and O. Naim. Dynamic Instrumentation of Threaded Applications. In *Proceedings of SEVENTH ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* May 1999.