RICE UNIVERSITY

Performance Modeling and Prediction for Scientific Java Applications

by

Rui Zhang

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE Master of Science

APPROVED, THESIS COMMITTEE:

Ken Kennedy, Co-chair
Ann and John Doerr University Professor, Computer Science

Zoran Budimlić, Co-chair
Research Scientist, Computer Science

John Mellor-Crummey
Associate Professor, Computer Science

Behnaam Aazhang
J.S. Abercrombie Professor, Electrical and Computer Engineering

Houston, Texas
February, 2006
Performance Modeling and Prediction for Scientific Java Applications

Rui Zhang

Abstract

This thesis develops an execution time model for predicting the performance of computation intensive scientific Java applications. Performance modeling can help in balancing the application workload of different problem sizes in a Grid environment. We propose a profiling based modeling method and introduce the concepts of point of predictability (PoP) and point of unpredictability (PoU) for performance prediction of Java programs. The point of predictability accounts for the volatile nature of the effects of JIT compilation on the execution time for small problems, while the point of unpredictability accounts for the effects of memory limitations and garbage collection on large problems. We present strategies to determine the point of predictability and point of unpredictability. Experimental results show that this model can accurately predict execution time for problem sizes between these two points which enables us to perform practical load balancing for scientific Java applications intended for execution on a Grid.
Acknowledgments

I would like to thank my advisors Ken Kennedy and Zoran Budimlić for their wisdom, patience and tremendous help, which guided me throughout my entire work. My other committee members, John Mellor-Crummey and Behnaam Aazhang, provided priceless guidance and comments on my work and thesis. I would also like to thank Gabriel Marin and Mackale Joyner for their help on the implementation and discussions on performance modeling related issues. Jan Hewitt from the Rice Cain Project provided tremendous guidance on my thesis writing. I would also like to thank my grandparents and my parents, who always support and encourage me on my academic endeavors.
## Contents

Abstract  iii

Acknowledgments  iv

List of Illustrations  vii

List of Tables  ix

1 **Introduction**  1

1.1 Motivation  1

1.2 Key Findings and Contribution  2

1.3 Modeling Process Overview  3

1.4 Related Work  3

2 **Preliminary Experiments**  14

2.1 Challenges for Java Performance Modeling  14

2.2 Java Application Performance Behavior  18

2.2.1 Experimental Applications and Platforms  18

2.2.2 Preliminary Experimental Results  18

3 **Point of Predictability**  26
Illustrations

1.1  Model Construction Process ........................................... 4

2.1  Execution Time of JavaParsek ........................................ 20
2.2  Execution Time of Smithwaterman ................................. 21
2.3  Parsek Regression Error Rate ........................................ 22
2.4  SmithWaterman Regression Error Rate ............................ 23

3.1  Load Balance of Parsek on Different Calibration Starting Point .... 27
3.2  Load Balance of Linpack on Different Calibration Starting Point .... 28
3.3  Load Balance of CholBench on Different Calibration Starting Point .... 29
3.4  Load Balance of QRBench on Different Calibration Starting Point .... 30
3.5  Load Balance of SVDCBench on Different Calibration Starting Point .... 31
3.6  Methods Compiled/Inlined of Parsek ................................. 32
3.7  Methods Compiled/Inlined of Linpack ............................... 33
3.8  Methods Compiled/Inlined of CholBench ............................ 34
3.9  Methods Compiled/Inlined of QRBench ............................... 35
3.10 Methods Compiled/Inlined of SVDCBench ....................... 35
3.11 Algorithm for Determining PoP .............................. 38

4.1 LUBench Memory Footprint Size under Different Max Heap Sizes .......................... 41
4.2 LUBench GC Percentage under Different Max Heap Sizes .................................. 42
4.3 LUBench GC Time under Different Max Heap Sizes ........................................ 43
4.4 GC Activity Comparison on Different JVM Implementations ............................. 44
4.5 CholBench Base/Peak Memory Footprint Size Prediction .................................. 45
4.6 QRBench Base/Peak Memory Footprint Size Prediction ...................................... 46
4.7 SVDCBench Base/Peak Memory Footprint Size Prediction ............................... 46

6.1 Methods Compiled/Inlined of LUBench .............................. 57
6.2 LUBench Base/Peak Memory Footprint Size Prediction of Sun 1.4.2
                  Windows ......................................................................... 58
6.3 LUBench Base/Peak Memory Footprint Size Prediction of Blackdown
                  1.4.2 Linux ........................................................................ 59
6.4 LUBench PoU Test ..................................................................... 59
6.5 LUBench Load Balance Test ...................................................... 60
Tables

2.1 Platforms .......................................................... 19
2.2 Parsek Profiling Data .............................................. 24
2.3 Linpack Profiling Data ............................................ 25
Chapter 1

Introduction

This thesis presents a strategy for modeling the execution time for scientific Java applications. The objective is to help achieve good load balance of distributed Java applications deployed on the Grid. Due to the Java execution model, performance modeling for Java applications is more complex than for native code. We discuss these new issues and introduce the concepts of point of predictability and point of unpredictability to summarize the complexities imposed by run-time compilation and garbage collection.

1.1 Motivation

The Grid [1] has rapidly emerged as an important computing platform. Different from conventional distributed computing, Grid computing focuses on large scale resource sharing in a highly distributed and heterogenous environment. Consequently, effectively balancing the workload on different computing nodes becomes critical to fully expose the potential computing power of the Grid.

Mandal, Kennedy, et al. [2] show that an in-advance heuristic workflow scheduling gives a better workspan than other existing scheduling strategies, given an accurate performance model of the application. The availability of an accurate performance model is
crucial for applying their technique.

Although a lot of research has been conducted for techniques of gathering performance information and modeling the performance of various types of applications, most of them focuses on native code applications [3, 4], not Java, which adopts a very different execution mechanism. The lack of an accurate performance model restricts the application of the in-advance heuristic workflow scheduling for Java applications. It is necessary to provide a good modeling technique for Java applications.

Our work is motivated by the above observations and focuses on developing techniques for modeling the execution time for Java applications. We present a practical grey-box strategy to model scientific Java applications to benefit load balancing of Java applications on Grid. We test our model by allocating a new benchmark application on three different platforms and present encouraging results.

1.2 Key Findings and Contribution

- Java virtual machine plays an important role in the performance of a Java application. Not only the implementation of the Java virtual machine, but also the set of parameters that one specific execution uses greatly affects the performance of a Java application and endows unique characteristics to Java applications compared with native code execution.

- Small problem size Java applications exhibit unpredictable execution time behavior. The main reason comes from the on-line optimization of byte code to native code,
which contributes a large percentage of the total execution time for small problem sizes. Some applications have a sharp performance change at large problem sizes due to the garbage collector interference.

- We introduce two new concepts, point of predictability and point of unpredictability, and the strategies to determine them. Point of predictability summarizes the volatile effects caused by on-line compilation. Point of unpredictability summarizes the effects of garbage collector.

- With point of predictability and point of unpredictability, we are able to isolate out a problem size zone where we are able to give a precise prediction, which enables our grey-box modeling technique to construct a performance model for achieving accurate load balance decisions.

1.3 Modeling Process Overview

Figure 1.1 shows the work flow graph of the process for constructing our performance model. The process is composed of four parts - time complexity model construction, point of predictability determination, point of unpredictability determination, and performance model construction. These techniques will be covered in the following chapters.

1.4 Related Work

Performance modeling has been studied extensively [3, 4, 5, 6, 7]. One area of interest is to capture how the execution time scales as problem size changes. A good scalability
Figure 1.1: Model Construction Process
model helps allocate jobs of different problem sizes effectively in a distributed computing environment. This thesis focuses on how to model the execution time as a function of its problem size for computation extensive scientific Java applications.

Existing techniques used by researchers for performance modeling include statistical prediction, profiling based prediction, simulation based prediction, static analysis based prediction, and manual model construction.

- **Statistical prediction:**

  Statistical prediction makes predictions based on statistical properties of past observations. It does not require detailed information about the target platform and the application. Usually a database of past observations needs to be kept. A prediction is made by choosing a set of past observations and calculating the prediction based on the statistical model. When new measurements are available, they will be added into the database.

  Iverson et al. [3] treat execution time as a random variable and predict it from past observations. They use the locality property of the performance data and make predictions by calculating the weighted average of a certain number of neighbors.

  The strength of statistical prediction is its ability to make predictions without detailed knowledge of the underlying hardware and the application. Theoretically it can be applied to any case where the statistical property holds. This makes statistical prediction applicable for situations where detailed performance information is not accessible. Another advantage of statistical prediction is that it can improve its pre-
diction accuracy naturally by accumulating more measurements, which makes tuning of the model easy and straightforward.

However, statistical prediction has several shortcomings. First, it treats the application and target platform essentially as black boxes and lacks the ability to reveal detailed information about performance. Second, the accuracy of statistical prediction highly depends on how typical the past observations are, the density of the past observations and how well the application’s performance fits the statistical model. It is not be able to predict accurately without sufficient past observations. In practice, a big database is not always available, neither is the ability to get new measurements. Third, statistical prediction is unsuitable for cross platform prediction. With detailed knowledge about the platform and the application concealed by its black-box approach, it is difficult for a statistical method to give a good cross-platform prediction.

Statistical prediction can be applied to Java applications directly, but its lack of accuracy and a required effort of maintaining a database do not make it the first choice when more performance information is accessible. Our model takes into consideration the unique properties of Java applications and uses profiling information to create parameterized models. Our model does not need to keep a large database and can give accurate prediction with less measurements than statistical model.

- Profiling based prediction:

  Profiling is a process of collecting information available at runtime by executing instrumented code or using hardware performance counters. It is a useful technique
to collect information difficult or impossible to get by static analysis. Instrumentation and hardware performance counters are two widely used techniques for profiling.

Instrumentation inserts sensor code at appropriate places in an application. When the instrumented code is executed, sensor code collects information about runtime activities of the application. Instrumentation usually incurs a big performance overhead for executing the additional sensor code.

Hardware performance counters are a set of hardware registers provided by most modern processors. Users can use these registers to record information such as number of cache misses, number of instructions issued, etc. Hardware performance counters are able to collect information on the hardware level with a minimum performance overhead.

Data collected via instrumentation or hardware performance counters is processed by analysis tools to extract the useful performance information and construct the model.

Snavely, et al. [5] map the application profiles to machine signature to construct the performance model for memory bounded applications. They use a set of benchmarks to collect the memory usage signature. By mapping the application’s profiles to the machine signature, they are able to model memory bounded applications accurately. Unfortunately, while effective, this model is confined to memory bounded applications.

Marin and Mellor-Crummey [6] developed a toolkit for cross platform prediction by profiling basic block execution frequency and analyzing memory hierarchy per-
formance via reuse distance analysis. Their model uses reuse distance analysis to predict the miss rate of L1 data cache, L2 data cache and TLB. By accumulating the delay from caches, TLB and the ideal execution time with no cache delays, their model is able to predict the execution time for in-order execution processors accurately. However, their techniques can not be applied to Java directly. If applied to Java, one possible way for their techniques is to treat JVM as the native application and Java applications as the input of the JVM. Unfortunately, JVM has a complex execution behavior and its performance depends highly on its input. These kinds of applications usually exhibit unpredictable behavior.

A part of our modeling process is based on profiling information. Our model is different from previous work because our model takes into account the unique characteristics of Java virtual machines which have not been investigated before. We use the actual execution time to summarize the complexities introduced from processor and memory hierarchy effects. Accurately modeling the interaction between the applications, the amount of parallelism used in the target platforms, the data access pattern and the memory hierarchy implementation presents a very large obstacle to accurate performance modeling. We believe the measurement of the actual execution time is an attractive way to summarize all the effect of these interactions and our model shows its effectiveness.

- Simulation based prediction:

Simulation is the process of executing applications on emulated target platforms in-
stead of on real platforms. It provides the flexibility to modify various architecture parameters to analyze performance on different platforms. Simulation allows for detailed execution profiling and facilitates a better understanding of performance related issues.

Various simulation techniques exist to meet different simulation objectives. One type of simulation is trace-driven simulation. Trace files are generated from profiling or analysis based on available information. Necessary information is kept in trace files. These files are processed by a simulator to simulate the activities on a specific platform. For example, to get the number of cache misses, a trace file can contain all data accesses to the memory. A cache simulator can give an accurate number of cache misses by simulating the cache behavior. One trace-driven cache simulator is DineroIII in WARTS [8].

Another type of simulation is the execution-driven simulation. This type of simulation virtually executes every step of the execution. RSIM [9] is an execution-driven simulator designed for shared-memory multiprocessor architectures.

Simulation can give very accurate results, but its performance overhead prohibits it from being practically applied to performance prediction of large applications. Spending a long time to simulate an application on one platform is not generally acceptable.

Another issue with simulation is that it does not reveal information about performance changes when the problem size scales. The result of a simulation is only
about a particular application in one specific setting. For these reasons we do not use simulation based technique in our modeling process.

- Static Analysis:

Static analysis derives information by analyzing the hardware architecture and the application statically [10, 11]. It does not require actually running the application and therefore can be applied early in development or in cases where actual execution of the application is not practical. Static analysis is very useful in detecting rare conditions that are difficult to detect in actual execution. It can prove some properties of an application that are difficult to do by run time data collection. For example, static analysis is able to tell that some condition will always be true, which a run time data collection can hardly prove. Although static analysis is quite useful at discovering static information, it is not effective at revealing run time information. Usually, it is used together with other techniques such as profiling and manual model construction.

- Manual Model Construction:

Manual model construction relies on a human expert to build the performance model manually [12, 13]. Necessary components for such a model construction are expert knowledge about model construction techniques, the application and the underlying hardware. By introducing the human into the process, this technique can produce a very accurate model. Moreover, it is able to model those cases that are very difficult,
if not impossible, for automatic model construction, like applications with irregular behavior. The shortcoming of this method is that it involves human interaction, which is not always desirable and can be very costly and time consuming.

Besides the one mentioned above developed by Marin and Mellor-Crummey, different performance analyzing toolkits have been developed. PACE [7, 14] developed by Kerbyson, et al. is one toolkit for predicting and analyzing performance issues in parallel systems. PACE provides a set of tools to help users to construct the model and analyze the behavior of a certain parallel application. But it does not have a systematic way of predicting the execution time when problem size scales. Adve et al. [15] develop an environment for end-to-end performance modeling of complex parallel and distributed systems. They suggest modeling the components of the target system using different modeling paradigms such as analysis, simulation, or direct measurement but do not give a specific method for modeling each component’s performance. Pablo [16] developed by University of Illinois Pablo Research Group is one performance analysis tool which eases the work of instrumentation, data collection, performance data visualization. But this tool still lacks the ability to predict the performance of different problem sizes. Paradyn [17] is a performance analysis tool for parallel and distributed programs. It provides a dynamic instrumentation tool and can provide precise performance data down to the procedure and statement level. There are other toolkits developed to meet different performance analysis requirements such as Falcon [18], VPPB [19].

However, performance behavior of Java applications is not as well studied. Shuf, et
al. [20] study the memory behavior of Java applications based on instrumentation. They investigated SPECjvm98 [21] and observed that array accesses are much more than accesses to method table. They also observed some opportunities of using prefetching to reduce cache misses. Rajan [22] studies the cache performance of SPECjvm98 on LaTTe [23]. He studies the performance of JVM in different phases - class loading, execution and garbage collection for different cache settings. Hsieh, et al. [24] investigate the cache and branch performance issues in Java applications. They compared the execution of Java applications among three types of execution - interpreted execution, JIT execution and C/C++ equivalent version execution - for data cache, instruction cache, and branch prediction buffer performance. Their data gives some insights into Java execution. For example, the interpretation mode incurs a lot of compulsory cache misses because the interpreter needs to set up new data structures often. Romer, et al. [25] examine four interpreters on different micro benchmarks and programs. They showed that the performance of an interpreter is not solely determined by the frequently executed command dispatch loop, but also by other factors like the expressiveness of the virtual command set and how effectively the set is used. Eeckhout et al. [26] investigate the interaction between Java program and virtual machines at the micro architecture level. Through their experiments with SPECjvm98 [21] and SPECjbb2000 [27], they revealed which part of activities on the micro architecture level is due to Java virtual machine, the application itself or the input to the application.

The remainder of this thesis is organized as follows. Chapter 2 discusses some unique issues Java performance modeling faces as opposed to native code model construction and
shows some preliminary experimental results and analysis, forming the guidelines for point of predictability and point of unpredictability. Chapter 3 defines point of predictability and presents a technique to estimate it. Similar to chapter 3, chapter 4 defines point of unpredictability and the technique to determine it. Chapter 5 describes the construction of the time complexity model and the performance model. Chapter 6 evaluates the efficiency of the model on resource allocation.
Chapter 2

Preliminary Experiments

Performance modeling in general is a complex problem because almost every aspect of a computing system affects the performance of an application. Compared with performance modeling of native code execution, performance of Java applications is even more complex because of the unique complexities Java virtual machine brings in. This chapter shows some experimental results of the performance behavior of several scientific Java applications and presents the challenges to performance modeling of Java applications.

2.1 Challenges for Java Performance Modeling

For general performance modeling, building a complete performance model is impractical, if not impossible. Factors such as processor architecture, memory hierarchy, storage system, workload of the target system, operating system, network conditions, program organization, etc. all affect the performance in some way. A complete performance model would be a function of all of these elements, but because of lack of understanding about how all these elements interact to each other and affect the performance model or the overwhelmingly large amount of data to be processed to get a detailed image, a complete performance model would be difficult and impractical to build.
Java brings in some unique characteristics to code execution. Instead of being compiled into native code, Java programs are compiled into platform neutral bytecode first. Bytecode cannot be executed on one computer directly; it is executed instead by an instance of Java virtual machine on the target platform.

To add to the complexity, there are different implementations of Java virtual machines available, each with unique run-time execution pattern. Although Java virtual machine implementations must comply with the Java virtual machine specification, the underlying implementations differ from each other and have different performance behavior. Three major types of JVM implementations have existed since the first JVM was launched - interpreted, JIT and mixed mode execution. The most widespread type of Java virtual machine now is mixed mode execution. Sun’s Hotspot [28] virtual machine is one of the mixed mode JVMs and is the primary target JVM for experiments in this thesis.

Typically a mixed mode Java virtual machine has one or more compilers, an execution engine and a garbage collector. A representative execution scenario over a mixed mode JVM involves code execution, class loading, interpretation, profiling, hot methods detection, compilation and garbage collection. During execution, JVM starts with interpreting the bytecode. At the same time JVM collects necessary profiling information to determine whether a method is hot enough and should be compiled. Methods determined as hotspots are sent to the compilation thread and compiled into native code. Future execution of these methods will switch from interpreted mode to native mode execution. Under some situation, JVM may decide to de-optimize or re-optimize some compiled code. During the
execution garbage collector may interfere anytime when the JVM decides the heap needs to be cleaned up.

Because all of the above activities happen at runtime, the execution time of a Java application includes all the time spent on them. The interaction among different components has impact on the performance as well. This is dramatically different from a typical native code execution, where the execution is solely determined by the application itself.

The fact that the execution of a Java application has to compete with other components for computing resources affects performance modeling of Java applications. Simply considering a Java application itself can no longer give an accurate image of its execution.

One important area of performance modeling is memory hierarchy, especially cache performance modeling. Modern cache implementations work in a way that is highly correlated with the data access pattern. A lot of research has been done in modeling the cache performance for native code execution. Java’s execution model makes cache analysis more complex. First, components of JVM like the interpreter, compiler, GC, profiler, etc. all compete for cache. Consequently, cache activity from the application execution is mixed with other JVM components and difficult to identify. Second, methods are executed in both the interpreted and native mode. Some methods are compiled into binary code at runtime. Optimizations applied during the compilation may change the memory access pattern and invalidate the previous result of the cache analysis before the compilation. Considering the fact that an individual method in a Java application may be compiled more than once, to find a consistent way of modeling the cache performance is rather a complex issue. Third,
memory space on the heap is handled by both the Java application and the garbage collector. Some garbage collection algorithms move objects on the heap during collection and update memory references accordingly, therefore the data access pattern from the Java program may change independently of the application. Finally, because register allocation has not been performed yet in bytecode, data accesses that will go to memory instead of registers are not clear at the class file level.

A satisfactory cache performance analysis of Java applications needs to solve the above issues. For example, reuse distance analysis needs to collect the reuse distance information via instrumentation and profiling. For Java, neither instrumenting the Java source code nor instrumenting the bytecode is a feasible solution to collect necessary information for reuse analysis because neither of them can reveal the true memory access pattern. Moreover, cache activities in JVM are a mixture of activities from the application and different components. The memory access pattern from the application is blurred. How to profile and what to profile stay as questions still at this phase. In this work, we choose to construct our performance model without an advanced cache performance analysis and focus on the effectiveness of using execution time and regression analysis. We believe, given that most scientific applications spend most of time on repeating computations and have regular data access patterns, an observation based grey-box modeling technique can exhibit most of the complexities the cache system imposes. Our speculation is validated by the experiments we conducted.
2.2 Java Application Performance Behavior

This section exhibits some experimental data showing the execution times of several scientific Java applications. We observe some interesting characteristics of Java applications. One is that although JVM has a complex execution mechanism, the execution time of large scale problems still exhibits a predictable pattern. On the other hand, for small problems, the execution time shows unpredictable behavior because of the complex activities happened in JVM.

2.2.1 Experimental Applications and Platforms

Six applications are investigated here - Parsek, Linpack, SmithWaterman, CholBench, QR-Bench and SVDCBench. Parsek [29] is a particle-in-cell simulation program to analyze the particle and energy behavior. Linpack is a linear algebra benchmark application. SmithWaterman is a database search application in biology. CholBench, QRbench and SVDCBench are different benchmark applications from the object-oriented linear algebra library Owlpack [29].

Platforms used in the experiments are listed in table 2.1.

Java virtual machine used include Sun Java 1.4.2, Blackdown 1.4.2, and IBM Java 1.3.1.

2.2.2 Preliminary Experimental Results

Figure 2.1 and 2.2 show how execution time changes for different problem sizes for Parsek and Smithwaterman. The pattern we observed on these figures encouraged us to use the
<table>
<thead>
<tr>
<th>Platform</th>
<th>CPU</th>
<th>Operating System</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Intel Pentium III 800MHz</td>
<td>Windows XP SP1</td>
<td>512MB</td>
</tr>
<tr>
<td>2</td>
<td>AMD Athlon 800MHz</td>
<td>Windows XP SP1</td>
<td>256MB</td>
</tr>
<tr>
<td>3</td>
<td>Itanium 2</td>
<td>Linux</td>
<td>8GB</td>
</tr>
<tr>
<td>4</td>
<td>Pentium III 450MHz</td>
<td>Windows XP SP1</td>
<td>256MB</td>
</tr>
<tr>
<td>5</td>
<td>Pentium M 1.86GHz</td>
<td>Windows XP SP2</td>
<td>768MB</td>
</tr>
<tr>
<td>6</td>
<td>Opteron 1.6GHz</td>
<td>Linux</td>
<td>8GB</td>
</tr>
</tbody>
</table>

Table 2.1: Platforms

regression modeling technique.

Figure 2.1 shows the execution time of Parsek from particle size 10 to 1,000,000 on five different platforms. We can observe a regular pattern on all five curves for large problem sizes. By analyzing the source code, we know Parsek has a linear time complexity model. Its time complexity model matches all curves on figure 2.1 quite well. Figure 2.2 is the execution time of Smithwaterman for different problem sizes on different platform combinations. Similarly, Smithwaterman has a linear time complexity model when input is the cell number and the curves match the linear pattern well.

Figure 2.3 and figure 2.4 are the graphs of the regression error for Parsek and Smithwaterman respectively. The execution time of each application is fit onto their time complexity model. These figures depict how far the regression data is off from the actual execution time. One common characteristic can be observed on these figures is that for large problem
sizes the regression error rate is usually quite small. This suggests that at large scale, the time complexity model is able to describe the shape of the execution time curve well, even with the existence of other components in JVM.

On the other side, we observed irregular activities when problem sizes are small as well. The activities happen inside JVM at the beginning of an execution can account for this behavior to some extent. When JVM starts executing an application, interpretation, profiling, hotspot detection, compilation, etc. are mixed together at the beginning of the execution. Activities such as interpretation, profiling, hotspot detection, compilation that happen for a small problem size will happen for a large problem size as well. Therefore,
the behavior of the JVM in startup phase becomes irregular and difficult to predict but at a large scale the performance becomes more regular and predictable.

The following profiling information of the compilation time and number of methods compiled of each application gives a better understanding of the role of each component on execution time. Table 2.2 and table 2.3 are the data for Parsek and Linpack respectively. On each table, columns 1 is the problem size for each execution. Column 2 is the number of methods executed in interpreted mode in each execution. Column 3 is the number of methods executed in compiled mode in each execution. Column 4 is the time spent on GC of each execution measured in ticks. Column 5 is the time spent on compilation of
Figure 2.3: Parsek Regression Error Rate

each execution measured in ticks. Column 6 is the total execution time of each execution measured in ticks. One tick is 20ms.

One column deserves some attention here. It is the compilation time on different problem sizes. On table 2.2 and table 2.3, with some small variations, the time spent on compilation rises along with the increase of the problem size and when the problem size reach a certain point, the compilation time stabilizes at a certain value. This is an encouraging result for modeling Java considering of its complex execution mechanism. It suggests the time spent on detecting hotspot methods and compilation take approximately a constant amount of time for large problem sizes. For the same reason a more regular execution pattern is expected for large problem sizes. Another information conveyed from these graphs and
Figure 2.4: SmithWaterman Regression Error Rate

tables is that the behavior of garbage collection is not affecting the performance behavior in an unpredictable way, which suggests a model based on measurement of the execution time has the potential to summarize the time spent on garbage collection as well.

Based on the pattern we observe above, it is reasonable to treat large problem sizes and small problem sizes differently. We want to find the problem size which can summarize the complexities of small problem sizes. We name this problem size as the point of predictability. For one specific application, suppose the execution time on one platform for the size of point of predictability is $T$, all execution time of problem sizes beyond this point can be modeled as $T + \delta T$, where $\delta T$ is the regular part and what we focus to model.
<table>
<thead>
<tr>
<th>Particle number</th>
<th>Interpreted</th>
<th>Compiled</th>
<th>GC ticks</th>
<th>Compilation</th>
<th>Received Ticks</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>6</td>
<td>N/A</td>
<td>1</td>
<td>80</td>
<td>63</td>
</tr>
<tr>
<td>100</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>79</td>
<td>68</td>
</tr>
<tr>
<td>1000</td>
<td>23</td>
<td>6</td>
<td>1</td>
<td>91</td>
<td>90</td>
</tr>
<tr>
<td>10000</td>
<td>27</td>
<td>158</td>
<td>2</td>
<td>107</td>
<td>258</td>
</tr>
<tr>
<td>50000</td>
<td>92</td>
<td>1247</td>
<td>12</td>
<td>115</td>
<td>1454</td>
</tr>
<tr>
<td>100000</td>
<td>110</td>
<td>2457</td>
<td>32</td>
<td>106</td>
<td>2706</td>
</tr>
<tr>
<td>500000</td>
<td>119</td>
<td>12179</td>
<td>150</td>
<td>105</td>
<td>12575</td>
</tr>
<tr>
<td>1000000</td>
<td>116</td>
<td>24377</td>
<td>360</td>
<td>99</td>
<td>25019</td>
</tr>
</tbody>
</table>

Table 2.2 : Parsek Profiling Data

This way the problem to model a Java application is transformed to the problem of locating the problem size which can summarize the irregular behavior for small problem sizes and how to locate this problem size. Next chapter will present our method to determine this point.
<table>
<thead>
<tr>
<th>Matrix size</th>
<th>Interpreted</th>
<th>Compiled</th>
<th>GC ticks</th>
<th>Compilation</th>
<th>Received Ticks</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>11</td>
<td>1</td>
<td>N/A</td>
<td>23</td>
<td>40</td>
</tr>
<tr>
<td>200</td>
<td>8</td>
<td>4</td>
<td>N/A</td>
<td>29</td>
<td>58</td>
</tr>
<tr>
<td>300</td>
<td>14</td>
<td>24</td>
<td>1</td>
<td>42</td>
<td>82</td>
</tr>
<tr>
<td>400</td>
<td>12</td>
<td>87</td>
<td>1</td>
<td>45</td>
<td>150</td>
</tr>
<tr>
<td>500</td>
<td>13</td>
<td>191</td>
<td>5</td>
<td>47</td>
<td>247</td>
</tr>
<tr>
<td>600</td>
<td>15</td>
<td>332</td>
<td>5</td>
<td>28</td>
<td>397</td>
</tr>
<tr>
<td>700</td>
<td>9</td>
<td>549</td>
<td>10</td>
<td>35</td>
<td>610</td>
</tr>
<tr>
<td>800</td>
<td>14</td>
<td>820</td>
<td>8</td>
<td>36</td>
<td>896</td>
</tr>
<tr>
<td>900</td>
<td>13</td>
<td>1186</td>
<td>14</td>
<td>34</td>
<td>1256</td>
</tr>
<tr>
<td>1000</td>
<td>12</td>
<td>1651</td>
<td>16</td>
<td>29</td>
<td>1721</td>
</tr>
<tr>
<td>1100</td>
<td>10</td>
<td>2186</td>
<td>27</td>
<td>32</td>
<td>2265</td>
</tr>
<tr>
<td>1200</td>
<td>15</td>
<td>2839</td>
<td>24</td>
<td>31</td>
<td>2921</td>
</tr>
<tr>
<td>1300</td>
<td>8</td>
<td>3599</td>
<td>28</td>
<td>29</td>
<td>3685</td>
</tr>
<tr>
<td>1400</td>
<td>16</td>
<td>4488</td>
<td>32</td>
<td>35</td>
<td>4577</td>
</tr>
<tr>
<td>1500</td>
<td>47</td>
<td>5526</td>
<td>35</td>
<td>66</td>
<td>5651</td>
</tr>
<tr>
<td>1600</td>
<td>12</td>
<td>3372</td>
<td>33</td>
<td>30</td>
<td>3467</td>
</tr>
<tr>
<td>1700</td>
<td>17</td>
<td>8015</td>
<td>42</td>
<td>35</td>
<td>8119</td>
</tr>
<tr>
<td>1800</td>
<td>16</td>
<td>9490</td>
<td>39</td>
<td>33</td>
<td>9596</td>
</tr>
<tr>
<td>1900</td>
<td>14</td>
<td>11198</td>
<td>44</td>
<td>40</td>
<td>11311</td>
</tr>
<tr>
<td>2000</td>
<td>12</td>
<td>12089</td>
<td>50</td>
<td>47</td>
<td>12205</td>
</tr>
</tbody>
</table>

Table 2.3: Linpack Profiling Data
Chapter 3

Point of Predictability

Point of predictability is the concept we introduce to summarize the volatile behavior of runtime compilation of Java applications. For a mixed mode JVM, the mixture of interpretation, compilation and native code execution complicates program behavior for small problems and makes an accurate prediction impractical. However, through our experiments, we discover that there exists a problem size that is able to summarize these complex effects and enables a practically good prediction for load balancing beyond this point. We define such a problem size as point of predictability and present its concept in this chapter. We also provide a strategy to determine the point of predictability.

3.1 Definition

Chapter 2 shows the performance behavior of Parsek and Smithwaterman. The data in chapter 2 shows the problem from the execution time perspective. While accurate execution time prediction would unarguably be useful for achieving good load balancing, resource allocation does not require it. Accurate relative performance prediction is sufficient. Our goal is to find a practical modeling technique to help with load balancing of Java workload, therefore it is helpful to view how different predictions affect the result of load balancing.
Figure 3.1: Load Balance of Parsek on Different Calibration Starting Point

Based on the observation that time complexity models can describe the execution time curve well at large scale, we want to see how well a load balance can be achieved if we choose different problem sizes to calibrate the regression model.

The following set of experiments was conducted to obtain an insight into the performance of scientific Java applications and the applicability of the regression model to the problem of application load balancing. We performed a large number of experiments in the following way: we pick a set of problem sizes to time the calibration runs, created a regression model based on those calibration runs, and predicted the execution time for given problem sizes using the regression model. Then we scheduled a set of problems on three machines using the predicted performance for maximizing the load balance. After that, we
Figure 3.2: Load Balance of Linpack on Different Calibration Starting Point

ran those problems on the machines and measured the actual load balance, to discover how
the achieved load balance depends on the problems sizes used for calibration.

We collected data for five applications - Parsek, Linpack, CholBench, QR Bench, SVD-
CBench. Load balancing results of them are shown on figure 3.1, 3.2, 3.3, 3.4 and 3.5. The
x-axis on each graph is the problem size where we start collecting calibration data. The
y-axis is the load balance index achieved for the corresponding calibration problem set.
The load balance index is the CPU utilization percentage as defined below.

Suppose the computation platform set is $C = \{c_1, c_2, \ldots, c_m\}$. The job set that will be
delivered onto these platforms is $J = \{j_1, j_2, \ldots, j_n\}$. For a certain workload decision, the
set of the execution time on each platform is $S = \{s_1, s_2, \ldots, s_m\}$. Let $s_{\text{max}} = \max(S)$,
then the CPU utilization percentage $P = \sum_{k=1}^{m} \frac{x_k}{x_{\text{max}}}$. Based on this definition, the perfect load balance's coefficient is 1. The smaller the coefficient is, the worse the load balance achieved. The smallest percentage is $\frac{1}{n}$ when the platform set contains $n$ platforms.

On each one of figure 3.1, 3.2, 3.3, 3.4 and 3.5, there are three curves. The "Predicted" curve stands for the predicted load balance index based on the predicted execution times. The "Actual" curve stands for the load balance actually achieved after allocating jobs onto each platform based on the prediction. The "Frequency based" curve stands for the actual load balance index achieved by using CPU frequency based prediction. For each application, 30 randomly generated problem sizes are used for load balancing. These generated problem sizes are large compared with the problem sizes used as calibration set.
Figure 3.4: Load Balance of QRBench on Different Calibration Starting Point

Let us look at figure 3.1 as an example. This figure shows the average CPU usage for three different machines, on which the load balancing algorithm scheduled 30 instances of the Parsek application of different problem sizes ranging from 200,000 to 1 million particles, using the predicted performance from the regression analysis calibrated by the test runs on the x-axis.

For the curve using predicted value for balancing, we expect its load balance to be close to 1. While perfect load balance cannot be achieved most of the time due to the set partitioning problem. For the curve using actual value, the closer it is to 1, the better the prediction is. CPU frequency based prediction is chosen as a relatively simple modeling technique to compare with. CPU frequency based prediction is derived with only information of the
CPU frequency and the time complexity model of the application. It assumes execution time scales with the time complexity model and the inverse of the CPU frequency. One example is for CPU frequency $f_1$, $f_2$ and time complexity model $O(n)$, the execution time on the first platform is $\frac{f_2}{f_1}$ times the execution time on the second platform.

The algorithm for load balancing used in these experiments is a straightforward greedy algorithm. In the process of allocating a new job, the algorithm always selects the platform which has the shortest job queue at that moment to assign the new job. To improve the allocation result, we repeat this process for a certain amount of times by shuffling the order of jobs and select the best one as the load balance achieved.

We observed a common pattern on all of these figures. On the actual curve, along with
Figure 3.6: Methods Compiled/Inlined of Parsek

As the increase of the problem size we start collecting calibration data, balance index fluctuates at first and then stabilize at a high value. The values where the actual curves stabilize are all better than the corresponding results utilizing CPU frequency based prediction.

This pattern implies that load balance index will achieve a good result when the problem size where to start collecting calibration set reaches a certain point. Beyond this point, the index of load balance stabilizes at a high point. Point of predictability is based on this observation that beyond one problem size a good load balance can be achieved by regression analysis. We call the first problem size on the actual curve that enters the stable phase the point of predictability.
3.2 Determining Point of Predictability

Although point of predictability has the nice property of enabling regression analysis prediction, the way to determine its size is still unclear. To get point of predictability, one might attempt to get it by drawing a graph like figure 3.1, 3.2, 3.3, 3.4 and 3.5. Unfortunately, it requires to run many instances of the problem with actual problem sizes, which makes the method impractical.

Therefore, in order to find a way to determine PoP, we attempted to discover factors highly correlated with it. Fortunately, through our experiments we find that the number of methods compiled by the JVM is highly correlated with PoP. Consequently, the problem of determining PoP can be transformed to the problem of finding the problem size that JVM
Figure 3.8: Methods Compiled/Inlined of CholBench

has compiled most possible hotspots.

Since volatile behaviour for small problems is closely related to on-line compilation, it is natural to see if the number of methods compiled is highly correlated with PoP. We collect the number of methods compiled on different problem sizes to examine if the number of methods compiled is a good indicator. We find if simply using the output by turning on -XX:+PrintCompilation for Sun’s JVM, the result is not satisfactory. The reason is Sun’s JVM applies method inlining extensively but the output from option -XX:+PrintCompilation does not report those methods that have been inlined, which makes the number of methods compiled not a stable indicator.

Therefore, we test if the sum of the methods compiled and the number of methods
Figure 3.9: Methods Compiled/Inlined of QR Bench

Figure 3.10: Methods Compiled/Inlined of SVDC Bench
inlined can work well. We find Sun's Hotspot JVM does not report how many methods are inlined. In order to get this information, we compiled the source code to get java_g. Using java_g, we are able to get the inlining information we need. Combined with the methods reported as compiled, we calculate the sum of the methods compiled or inlined. The number of method compiled or inlined exhibits a correlation with PoP.

On figure 3.6, 3.7, 3.8, 3.9, 3.10, the diamond/blue curves stand for the number of methods compiled or inlined for the corresponding problem size. These experiment datum back up the observation that PoP is highly correlated with the problem size where almost all compilation has been finished. In other words, it is the problem size that enters the rightmost highest plateau on the diamond/blue curve.

In the process of searching for the PoP, we want to avoid accepting one of the lower plateaus to the left of PoP as a solution. To achieve this goal, we use another factor, the compilation percentage in the total execution time, to help locate PoP on the diamond/blue curve. We use the compilation percentage curve to screen out problem sizes that the search might be stuck in.

The compilation percentage is depicted as the square/pink curves on figure 3.6, 3.7, 3.8, 3.9, 3.10. By discarding problem sizes with a large compilation percentage, we have a good chance of keeping the search out of lower level plateaus.

By examining the results from our application suite, we have reached an empirical value of 30%. In other words, once the time JVM spends compiling code falls below 30%, we can be fairly certain that the PoP lies on the first plateau right of that point.
Formally, the algorithm to determine the PoP is shown on figure 3.11.

The algorithm starts with a small problem size $s_0$, which is the problem size where the search starts. $p$ is the incremental factor for increasing the problem size. For each problem size, the algorithm collects the number of the methods compiled, inlined and the percentage of compilation to the total execution time, $p_c$. Problem sizes whose $p_c$ is less than a threshold, $\theta$, are discarded. After $p_c$ is greater than $\theta$, the algorithm starts searching for the first plateau. The algorithm computes the standard deviation of the number of methods compiled or inlined for several consecutive points. When the standard deviation is less than some threshold, $\gamma$, the algorithm stops and returns the first problem size on the plateau as the point of predictability. In our experiments, $s_0$ is 10, $p$ is 10%, $\theta$ is empirically set to 30% and $\gamma$ is empirically set to 0.75.
inputs: $IN$ (initial problem size $s_0$)

(problem size incremental percentage $p$)

(compilation percentage threshold $\theta$)

(standard deviation threshold $\gamma$)

(list $L$)

(size of $L$ $n$)

outputs: $OUT$ (point of predictability)

begin

\[ s = s_0 \]

while $p_c > \theta$

\[ p_c = \text{compilation\_percentage}(s) \]

\[ s = s \times (1 + p) \]

endwhile

for $i = 0; i < n; i++$

\[ L\_\text{add}(\text{compiled\_method\_num}(s), s) \]

\[ s = s \times (1 + p) \]

endfor

while $STD(L) > \gamma$

\[ L\_\text{remove\_first}() \]

\[ s = s \times (1 + p) \]

\[ L\_\text{add}(\text{compiled\_method\_num}(s), s) \]

endwhile

return $L\_\text{first}.s$

end

Figure 3.11: Algorithm for Determining PoP
Chapter 4

Point of Unpredictability

Point of unpredictability is another concept we introduce to help model the performance of Java applications. The reason for introducing it is that the garbage collector interacts with the heap in an unpredictable way for some applications when the problem size reaches a certain value. We motivate the necessity of this concept in this chapter and present a technique for determining the point of unpredictability by predicting the memory footprint size.

4.1 Definition

Figure 4.1 shows the garbage collection activities of LUBench for the same problem size under two different maximum heap sizes. The diamond/blue curve is the case when the maximum heap size is 32MB. The square/pink curve is the case when the maximum heap size is 128MB. We notice that the garbage collection happens much more often when the maximum heap size is 32MB. As a consequence, the total execution time when the heap size is 32M is more than twice longer than the case when the heap size is 128MB.

Figure 4.2 and 4.3 show this problem from different perspectives. Figure 4.2 shows how the percentage of garbage collection over the total execution time changes for different
problem sizes under different maximum heap sizes. Figure 4.3 shows how the time spent on garbage collection changes for different problem sizes under different maximum heap sizes. Both of these two figures share a similar pattern. For small problem sizes, the three curves on each graph are almost the same, but they diverge from each other at some point.

The reason for these sharp changes lies in the interaction between the heap and the application’s memory requirement. When the application’s memory footprint size reaches the maximum heap size, the total available heap memory approaches zero. The garbage collector can not keep up with the garbage production and has to collect the heap more often. There are two possible scenarios when the maximum heap size is reached. One is when the footprint size of the application has space to be compressed and garbage collector will be able to clean enough room for the application to continue execution. We call this ”heap thrashing”. Another one is when there is not enough space to clean any more regardless of how hard the garbage collector works. In this case, the application runs out of heap space and reports out of memory error. Because the time to do garbage collection is included in the execution time, if the garbage collector has to work more often, the total execution time will be increased dramatically.

Two considerations motivate us to find a way to predict when these two scenarios will happen. First, the dramatic change for execution time is important for execution time prediction because the different patterns need different ways to model. Second, knowing the points when these scenarios will happen can provide useful information for selecting suitable resources and JVM parameters. Due to a lot of time is spent on garbage collection,
it is not desirable to execute a Java application under "heap thrashing" conditions.

In order to predict when an application enters "heap thrashing" mode, we develop a method to predict the memory footprint size of the application on heap and derived a strategy to accurately predict the critical problem size entering "heap thrashing" mode accordingly. We define the minimum problem size that leads the problem to enter the "heap thrashing" mode as point of unpredictability. Point of unpredictability is a function of the application, maximum heap size and garbage collection implementation. Next section gives the related experiment data and our strategy to determine PoU.

Figure 4.1: LUBench Memory Footprint Size under Different Max Heap Sizes
Figure 4.2: LUBench GC Percentage under Different Max Heap Sizes

4.2 Determining Point of Unpredictability

We introduce two terms to help determine the point of unpredictability. One is the base footprint (BF), the other one is peak footprint (PF). BF is the minimum memory footprint size one application occupies for the execution of one problem size. Essentially it represents the heap space occupied by the application that can not be collected. PF is the maximum memory footprint size an application can reach for one problem size given sufficient heap space. Sufficient here means that none of the garbage collection activities is triggered because of the heap size limit.

In this thesis, we investigate the behavior of the default garbage collector used in Sun’s hotspot Java virtual machine. It is a generational garbage collector [30]. A generational
garbage collector utilizes the statistical property of heap object lifespan; statistically most objects on heap exist for a short period of time while a small number of objects that are left have longer lives. Generational garbage collectors divide all objects into different generations and collect younger objects more often than older objects. This method effectively reduces the total time spent on garbage collection. Most garbage collection events are for one generation. Full garbage collections, which scan through all objects on heap and collect all unreachable objects, happen much less often.

Therefore for a generational garbage collector, BF is achieved after a full garbage collection is performed; PF is possible to be got by any garbage collection. We monitor each execution and gather information of the garbage collection activities. The information
Figure 4.4: GC Activity Comparison on Different JVM Implementations

of BF and PF is extracted by searching for the largest memory footprint size after a full garbage collection and before each garbage collection respectively.

The memory footprint size is the result of the interaction between the data allocation pattern of the application and the garbage collector. We conduct some experiments using regression analysis and find we are able to predict the footprint pattern well. We use the maximum base footprint size and peak footprint size as the calibration data for regression analysis. Figure 4.5, 4.6 and 4.7 show the actual BF and PF and predicted BF and PF on different problem sizes of CholBench, QRBench and SVDCBench respectively. These experiments validate that regression analysis is able to give an accurate prediction of BF and PF.
Figure 4.5: CholBench Base/Peak Memory Footprint Size Prediction

With the ability of predicting the BF and PF accurately, we establish our method of determining PoU based on the prediction. Our method works as follows.

First we select a set of problem sizes to gather calibration data for predicting BF and PF. Then we collect GC profiling data and extract the BF and PF information for each problem size. After that we use regression analysis to find the best fit functions for both BF and PF, $f_{BF}(problemsize)$ and $f_{PF}(problemsize)$. After having the fit functions, we calculate PoU by solving the equation $f_{PF}(x) = MaximumHeapSize$, where $MaximumHeapSize$ is the maximum heap size set by the user.

One thing to note is PoU does not always exist for a given application. This happens when the application does not have a "heap thrashing" mode, or in other words, BF = PF.
Figure 4.6: QR Bench Base/Peak Memory Footprint Size Prediction

Figure 4.7: SVD Bench Base/Peak Memory Footprint Size Prediction
For applications with this property, the executions run out of memory directly when the problem sizes exceed the maximum heap size. The method stated in this chapter is still able to predict this situation. When the analysis detects that BF and PF are the same, it solves the equation $f_{BF}(x) = MaximumHeapSize$ to get the problem size where the execution will run out of memory.

Another thing to note is that PoU is dependent on the underlying implementation of the Java virtual machine. For example, figure 4.4 shows the GC activities of the same problem size 678 of LUBench on two different JVM implementations. One is on Sun’s 1.4.2 Hotspot JVM on Windows. The other one is Blackdown 1.4.2 on Linux. These two curves exhibit different GC patterns. Therefore they have different BF and PF sizes. But for each implementation, our method can predict the size of PF and BF accurately and predict PoU accurately.
Chapter 5

Performance Model Construction

The previous two chapters establish two novel concepts PoP and PoU and present our strategies to determine them. After determining these two points, we have a problem size zone in between of PoP and PoU, where the performance pattern is less affected by JVM and suitable for prediction. This chapter is about how to build the performance model to predict problems between PoP and PoU. Briefly speaking, our strategy is to profile the application, derive its time complexity model and then fit execution time data on the time complexity model to generate the performance model. The following sections describe these concepts in detail.

5.1 Gathering Information for Building Time Complexity Model

A time complexity model is the characteristic of an application itself and independent from the target platform. It describes asymptotically how much computation steps the execution needs. Some application’s time complexity model is sensitive to the content of their input. One example is the quick sort algorithm. Given different sets of elements to be sorted, the time complexity model of the quick sort algorithm can be either $O(n^2)$ or $O(n)$. The performance modeling of this kind of applications needs to model the input’s content as
well and is beyond the consideration of this thesis.

Many ways exist for building a time complexity model, such as manual construction, static analysis or profiling. Manual construction takes advantage of human analytical ability and can build very accurate models. Its shortage is that it can not be automated and requires expert knowledge. Therefore the availability of manual model construction is limited and the whole process can be expensive and time consuming. Static analysis builds the model by analyzing the application statically. The lack of runtime information prohibits it from giving a good result when it is not capable of discovering enough information. On the other hand, profiling can partially solve the problem of static analysis for the access to runtime data of the application. We choose to use profiling based regression method to build the time complexity model in our work.

Profiling needs to insert sensor code at appropriate places to collect necessary data. For building the time complexity model, our attention is on control structures in the application.

In an extreme case that an application does not have any control structure, we can expect the application to run straight from the beginning to the end. This type of application has a very simple time complexity model - O(1). Control structures are the elements that complicate time complexity models. Control structures, such as branches and jumps, give an application the possibility to execute differently every time. Knowing how control structures work in an application is critical for building its time complexity model.

Therefore, our method of instrumentation is to insert sensor code to monitor the activities of control structures. At the Java source level, several types of control structures exist,
such as loop, if statement, exceptions, etc. Each of them plays its role in the time complexity model. Gathering information for all of them is an expensive procedure. Without losing generality, we collect information of the most important control structure, loops, in our experiment. Our method can be easily extended to more general cases if necessary.

For each loop in the application, we collect its taken time and not taken time for a set of different problem sizes. Then we perform regression analysis to find the best matched function for each loop.

We identify a loop by the call site chain starting from the entry point to the loop. A call site chain is the concatenation of all call sites from the entry point to the last method that contains the loop. For example, if method MAIN calls another method FOO, and there is a FOR loop in FOO, the call site chain to the FOR loop in FOO from MAIN will be MAIN-FOO. If a FOR loop is in method FOO and FOO is called from two different locations MOO1 and MOO2, the FOR loop called from these two locations are different because one key is MOO1-FOO and the other one is MOO2-FOO. This way we are able to distinguish a loop called on different paths. A context-aware loop identification system can give a better image of the execution. Next section is about how we instrument the Java source code and process profile data.

### 5.2 Instrumentation and Data Processing

The instrumentation is implemented on Sun's 1.5 javac compiler and JaMake [29] framework developed at Rice university.
We use a very straightforward instrumentation strategy in our experiments. The instrumentation algorithm is not a very time efficient one, but it works well enough because we only execute very small problem sizes to build the time complexity model. Ball and Larus have a more efficient profiling strategy published in [31]. Their strategy can easily be adopted into our framework.

Our instrumentation inserts code before each method call to push the current call site onto a stack. After each method call, code is appended to pop the call site location off the stack. Also for each loop in the application, code to count its taken times and not taken times is inserted at proper locations as well.

Before instrumentation, we apply another transformation which we call canonization. The purpose of canonization is to make the code more suitable for instrumentation. For example, in the case some method calls are embedded deep inside some statement, canonization moves these method calls out to be individual statements. Examples are provided below for canonization and instrumentation separately.

Before canonization:

\[ z = \text{foo}() + \text{goo}(); \]

After canonization:

\[ x = \text{foo}(); \]
\[ y = \text{goo}(); \]
\[ z = x + y; \]
Before instrumentation:

```plaintext
x = foo();
```

After instrumentation:

```plaintext
Counter.callSiteStack.push("key");

x = foo();

Counter.callSiteStack.pop();
```

Next is an example to illustrate what a loop is like after instrumentation.

Before instrumentation:

```plaintext
for (i = 0; i < n; i++) {
    x[i] = b[i];

    i++;
}
```

After instrumentation:

```plaintext
Counter.createOneCounter("key for loop taken");
Counter.createOneCounter("key for loop not taken");
Counter.addOneToCounter("key for loop not taken");

for (i = 0; i<n; i++) {
    Counter.addOneToCounter("key for loop taken");
}
```
\[ x[i] = b[i]; \]
\[ i++ ; \]

The instrumented code is executed on a set of small problem sizes to gather their loop taken and not taken counts. The data files collected are then processed by a regression analysis tool to find the best fit function of each loop. For a predictable application, we expect loops in it to follow some regular behavior.

The following is one set of data from Linpack for matrix size 100, 200, 300, 400, and 500.

Key: SomeKey

Value: [10000, 40000, 90000, 160000, 250000]

The process of determining the best fit function is a search process. We set a threshold to check if a function can fit well enough. The search tries each possible function in a predefined list. Once the threshold is met, the search stops and the best fit function at this moment is chosen as the function for this loop. This process is repeated for each loop in the application to find the function to fit each loop.

One issue in the fitting process is that the search space is confined by the function list provided. The function list contains a series of candidate functions. In the case that no function in the list can fit the loop behavior well enough, the fitting process fails to find
the best fit. While in practice most applications' time complexity models are polynomial functions, this case rarely happens actually. In our experiments, the list is composed by polynomial functions of different orders. If necessary, more candidate functions can be added.

With the best fit function for each loop, we are ready to build the performance model in the next section.

5.3 Performance Model Construction

The execution count gathered for each loop is the overall taken or not taken time of a specific loop. Without the execution time each loop takes, it is impossible to unify all loops into one overall accurate function. Fortunately the highest order of the unified function is sufficient for us to construct the polynomial function to describe the performance curve.

We scan through all loops to find the one whose fit function has the largest order. If the largest order is n, the basic performance model function will be \( f(x) = a_1x^n + a_2x^{n-1} + \ldots + a_nx + a_{n+1} \), where \( f(x) \) stands for the execution time when the problem size is \( x \).

To build the performance model of an application, a set of actual execution times are needed to materialize the coefficients in \( f(x) \). We choose a set of small problem sizes and collect the execution time of each one. The calibration problem sizes are chosen in the range beyond point of predictability. Then the set of execution times are fit onto \( f(x) \) to materialize each coefficient and generate the performance model.

From the experiments in previous chapters, we expect that the model function \( f(x) \) will
be able to make accurate load balance decisions. We will evaluate the effectiveness of this technique in the next chapter.
Chapter 6

Performance Model Evaluation

We further evaluate the effectiveness of our modeling method on a new application by allocating a set of problems onto three different platforms. The test is done on a computer with 2 AMD 1.6GHz Opteron processors, one desktop with AMD Athlon 800MHz and one notebook with Intel Pentium M 1.86GHz processor. The application used for the test is LUBench in Owlpack.

Following our modeling procedure, we first instrument the source code and collect data for building its time complexity model. The time complexity model we get for LUBench is $O(n^3)$.

Then we use the methods in this thesis to find the point of predictability and the point of unpredictability. The PoP is 215 for this problem, as shown in figure 6.1.

Let $ps$ stand for the problem size, from the data on figure 6.2 for Sun 1.4.2 JVM on Windows, the regression results for BF and PF are

$$BF = 0.06757 \times POWER(ps, 2) + 9.463 \times ps - 861$$

$$PF = 0.1387 \times POWER(ps, 2) + 23.21 \times ps - 1930.$$  

PoU is calculated for the case that the maximum heap size is 128MB. PoU is 885 for 1.4.2 windows version. PoU is 650 for Blackdown 1.4.2 implementation.
Figure 6.1: Methods Compiled/Inlined of LUBench

Figure 6.2, 6.3 and 6.4 evaluate the result of our predictions of BF, PF and PoU.

Figure 6.2 shows both the actual and predicted BF and PF of LUBench on Sun 1.4.2 JVM for Windows. Figure 6.3 shows both the predicted and actual BF and PF of LUBench on Blackdown 1.4.2 for Linux. On both figures the predicted curve and actual curve are very close, which means our predictions are accurate.

Figure 6.4 is the figure to evaluate the accuracy of the PoU prediction. It shows the execution time of LUBench on different problem sizes on Sun's Hotspot JVM for Windows and Blackdown for Linux when maximum heap size is 128MB and 512MB. For both JVM implementations, the 128MB curves diverge from the 512MB curve at the problem size we predicted, which proves our predictions of PoU are accurate.
Figure 6.2: LUBench Base/Peak Memory Footprint Size Prediction of Sun 1.4.2 Windows

As a final step, we test the effectiveness of using performance model to do load balancing. We generated a set of 30 problem sizes randomly in between of PoP and PoU. The load balancing index achieved from our modeling strategy is 0.942. The result by using CPU frequency for prediction is 0.691. The load balance result is satisfactory and proves the effectiveness of our modeling method.
Figure 6.3: LUBench Base/Peak Memory Footprint Size Prediction of Blackdown 1.4.2 Linux

Figure 6.4: LUBench PoU Test
Figure 6.5: LUBench Load Balance Test
Chapter 7

Future Work and Conclusions

7.1 Future Work

- A better strategy for doing memory hierarchy analysis for Java applications. Currently we measure the actual time to amortize the effect, which in some cases prevents a good prediction. A technique for modeling the memory hierarchy performance effectively would be very helpful.

- A better strategy for modeling garbage collection effects on the execution time. There are different garbage collection strategies available and each of them has different behavior. Understanding and modeling garbage collection behavior will give a better understanding of Java applications’ performance.

- We are considering adaptively tuning of the existing model when feedback information of the system is available.

7.2 Conclusions

Performance modeling of Java applications is a novel research area. The Java execution model introduces some unique complexities to Java applications compared with native code
applications.

We present an approach for performance modeling of Java applications that is practical and accurate. We introduce two novel concepts: point of predictability and point of unpredictability to summarize the volatile behavior of JVM for small and large problems. These two novel concepts help define a problem size range where the performance of a Java application can be accurately predicted. We also develop the strategies for determining PoP and PoU whose accuracy is evaluated by our experiments.

We use code instrumentation on very small problem sizes to construct an accurate time complexity model. We use point of predictability to create a set of calibrating runs that provide the data for constructing the performance model.

We have evaluated and validated our performance model by scheduling 30 jobs on three different machines to maximize the load balance. We achieved a load balance of 0.942 by using our strategy, which is an enormous improvement over a simple frequency-based scheduling strategy.
Bibliography


