Performance Modeling and Prediction for Scientific Java Applications

Rui Zhang, Zoran Budimlić, Ken Kennedy
Department of Computer Science
Rice University
Houston, Texas 77005
{ruizhang, zoran, ken}@rice.edu

Abstract

With the expansion of the Internet, the Grid has become a very attractive platform for scientific computation. Java, with a platform-independent execution model and a support for distributed computation built into the language, is an inviting choice for implementation of applications intended for Grid execution. Recent work at Rice University has shown that a reasonably accurate performance model combined with a load-balancing scheduling strategy can significantly improve the performance of a distributed application on a heterogeneous computing platform, such as the Grid. However, current performance modeling techniques are not suitable for Java applications, as the virtual machine execution model presents several difficulties: (1) a significant amount of wall time is spent on compilation in the beginning of the execution, (2) the virtual machine continuously profiles and recompiles the code during the execution, (3) garbage collection can interfere at any point in time and reorganize the data with unpredictable effects on memory hierarchy, (4) the application can spend more time garbage collecting than computing for certain applications and heap sizes and (5) small variations in virtual machine implementation can have a large impact on the application's behavior.

In this paper, we present a practical profile-based strategy for performance modeling of Java scientific applications intended for execution on the Grid. We introduce two novel concepts for the Java execution model: Point of Predictability (PoP) and Point of Unpredictability (PoU). PoP accounts for the volatile nature of the effects of the virtual machine on total execution time for small problem sizes. PoU accounts for the effects of garbage collection on certain applications that produce a lot of garbage and have a memory footprint that approaches the total heap size. We present an algorithm for determining PoP and PoU for Java applications, given the hardware platform, virtual machine and heap size. We also present a code-instrumentation-based mechanism for building the algorithm complexity model for a given application. We introduce a technique for calibrating this model using the PoP that is able to accurately predict the execution time of Java programs for problem sizes between PoP and PoU. Our preliminary experiments show that using these techniques can achieve load balancing with more than 90utilization.

1 Introduction

We present a strategy for modeling and predicting the execution time of scientific Java applications as a function of the problem size. The objective of this work is to achieve good load balance of distributed Java applications deployed on the Grid. We introduce the concepts of point of predictability and point of unpredictability in Java applications, and present our methodology for modeling and predicting the performance of Java programs.

1.1 Motivation

The Grid[7] 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 heterogeneous environment. Wisely balancing the workload on different computing nodes is critical to fully exposing the potential computing power of Grid.

Mandal, et al. [15] 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. An accurate performance model is crucial for the quality of the balancing result when using plan ahead scheduling strategies.

Java programming language is an attractive candidate for building applications for Grid because of its inherent platform independence and orientation toward network computing. Unfortunately, performance modeling of scientific Java applications has rarely been investigated, although performance modeling has been a research area for a long time. To facilitate the development of Java applications on the Grid, an accurate performance model for Java applications is important.

In this paper, we present a practical strategy to model scientific Java applications to improve load balancing of Java applications on the Grid.

1.2 Key Findings

Java virtual machine's implementation greatly affects the performance of a Java application and endows unique characteristics to Java applications compared with native code execution.

Java programs on small problems exhibit unpredictable execution time. Execution time for problems in a certain zone is more regular and predictable.

Some Java applications exhibit erratic behavior when the application footprint approaches the size of the heap, and the application is producing garbage at an increased rate.

We introduce two new concepts, point of predictability and point of unpredictability, and the strategies to determine them.

1.3 Related Work

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 of the application [12]. This strategy is able to make predictions without detailed knowledge of the underlying hardware and the application, and also to improve the accuracy by accumulating more measurements. However, it lacks the ability to reveal detailed information about performance, its accuracy highly depends on how typical the past observations are, and it is unsuitable for cross-platform prediction.

Profiling Based Prediction Instrumentation and hardware performance counters are two widely used techniques to get profiling information [16]. Morin and Mellor-Crummey [16] developed a toolkit for cross platform prediction by profiling basic block execution frequency and analyzing memory hierarchy performance via reuse distance analysis. Unfortunately, their techniques can not be applied to Java directly, since JVM has a complicated execution behavior and its performance depends highly on its input, making the behavior unpredictable.

Snavely, Carrington and Wolter[23] 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 machine signatures, they are able to model memory bounded applications accurately. Unfortunately, this model is confined to memory bounded applications.

Simulation Based Prediction Simulation is the process of executing applications on emulated target platforms instead of on real platforms, and it could be trace-driven [1], or execution-driven [18].Simulation can give very accurate results, but its performance overhead prohibits it from being practically applied to performance prediction of large applications. Another issue with simulation is that it does not reveal information about how the performance changes when problem size scales.

Static Analysis Static analysis derives information by analyzing the architecture of the application [8, 26]. This technique is very useful in detecting rare conditions or proving program properties that are difficult to achieve by actual execution. On the other hand, static analysis is weak at discovering some run time information. Performance prediction based solely on static analysis usually relies on some assumptions about the program properties that are not available. Because of this, static analysis is usually used in conjunction with other techniques.

Manual Model Construction Manual model construction relies on the programmer or expert to build the performance model manually [24, 10]. Expert knowledge of model construction, the application and underlying hardware is necessary. Introduction of the human into the process can result in a very accurate model, even for the programs that are very hard if not impossible to model automatically, such as irregular applications. Unfortunately, human interaction isn't always desirable or possible, and can be time consuming.

Besides the toolkit mentioned above developed by Marin and Mellor-Crummey, we should also note PACE developed by Kerbyson et al. [14, 13], Pablo developed by University of Illinois Pablo Research Group [20], ParaDyn [17], Falcon[9], VPPB[2], etc.

Unfortunately, none of these toolkits provide the functionality needed for predicting the performance of Java programs. They mostly deal with complied code modeling and prediction, and cannot handle complex interferences to program execution, such as just-in-time compilation and garbage collection.

Performance behavior of Java applications isn't as well studied. Shuf, Serrano, Gupta and Singh[22] studies the memory behavior of Java applications based on instrumentation. Rajan [19] studies the cache performance of SPECjvm98 on LaTTe. Hsieh, Conte, Johnson, Gyllenhaal and Hwu[11] investigate the cache and branch performance issues in Java applications. Romer, et al. [21] examine the performance four interpreters on different micro benchmarks and programs. Eeckhout, Georges and Bosschere[6] investigate the interaction behavior between Java program and virtual machines at the micro architecture level. While providing very useful insights into Java performance, none of these studies provides a performance modeling method for predicting performance of Java programs.

The remainder of this paper is organized as follows. Section 2 discusses some unique properties Java performance modeling faces as opposed to native code model construction and shows some preliminary experimental results and analysis, forming the guidelines for the concepts of point of predictability and point of unpredictability. Section 3 defines point of predictability and presents a technique to estimate the problem size for this point. Similarly, section 4 defines point of unpredictability and the technique to determine it. Section 5 describes the construction of the time complexity model and the performance model. Section 6 provides experimental results that evaluate the efficiency of our model on resource allocation problem for a collection of applications.

2 Preliminary Experiments

Performance modeling in general is a complex problem because almost every aspect of a computer system can affect the performance of an application. Performance of Java applications is even more complex because of the unique complexities Java virtual machine brings in.

2.1 Challenges for Java Performance Modeling

Even for performance modeling of native code, building a complete performance model is impractical. Factors like processor architecture, memory hierarchy, storage system, workload of the target system, operating system, network conditions and program organization will affect the performance model.

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.

A typical execution of a Java application on a mixed mode JVM involves class loading, interpretation, profiling, hot methods detection, compilation and garbage collection. All these activities compete for the CPU time and for cache. Furthermore, there are different implementations of Java virtual machines available, each with a unique run-time execution pattern.

Due to these predicaments, we believe that a traditional approach to performance modeling and prediction that involves static code analysis [8, 26] and reuse distance analysis [16] is impractical for Java applications. Instead, we choose to focus on the effectiveness of using instrumentation, profiling, execution time prediction and regression analysis to construct the model.

This section exhibits some experimental data of the execution time of several scientific Java applications on different platforms. We discover some interesting characteristics of Java applications. For example, although JVM has a complicated execution mechanism, execution time for large scale problems still exhibits a predictable pattern. On the other hand, for small problems, the execution time shows unpredictable behavior because of the complicated activities happening in JVM.

2.2 Experimental Environment

We collected a small suite of Scientific Java applications to drive our experimental efforts. This suite involves six applications: Parsek, Linpack, SmithWaterman, CholBench, QRBench and SVDCBench. Parsek is a particle simulation program to analyze the particle and energy behavior. Linpack is a linear algebra Java program, written in a traditional Fortran coding style. SmithWaterman is a database search application. CholBench, QRBench and SVDCBench are parts of the object-oriented linear algebra package called OwlPack [3].

Platforms used in the experiments are listed in table 1.

We have conducted our experiments on several Java virtual machines, including Sun Java 1.4.2, Blackdown 1.4.2, and IBM Java 1.3.1.

Platform	CPU	Operating System	Memory
1	Intel Pentium III 800MHz	Windows XP SP1	512MB
2	AMD Athlon 800MHz	Windows XP SP1	256MB
3	Itanium 2	Linux	8GB
4	Pentium III 450MHz	Windows XP SP1	256MB
5	Pentium M 1.86GHz	Windows XP SP2	768MB
6	Opteron 1.6GHz	Linux	8GB

Table 1. Platforms

2.3 JVM Execution Behavior

Figure 1 shows how execution time changes for different problem sizes for Parsek and SmithWaterman.

We conducted a series of preliminary experiments measuring total execution times of the applications, while varying the problem size. The purpose of these experiments was to gain an insight into the execution behavior of scientific Java programs, hoping to discover some regularity, despite the complications of the JVM execution described above.

Graph (a) in figure 1 shows the execution time of Parsek from particle size 10 to particle size 1,000,000 on five different platforms. A regular pattern is observed on all five

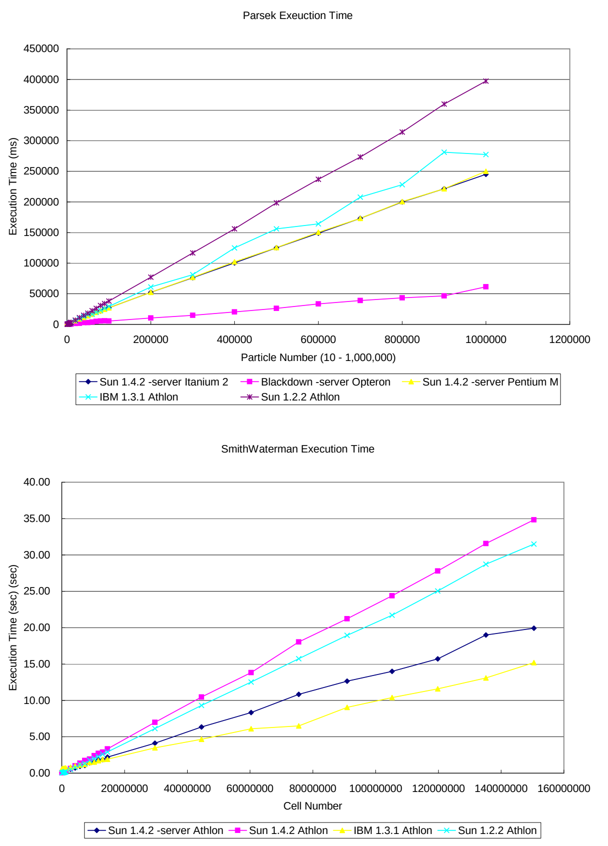

Figure 1. Execution Time of Parsek and SmithWaterman on Different Platforms

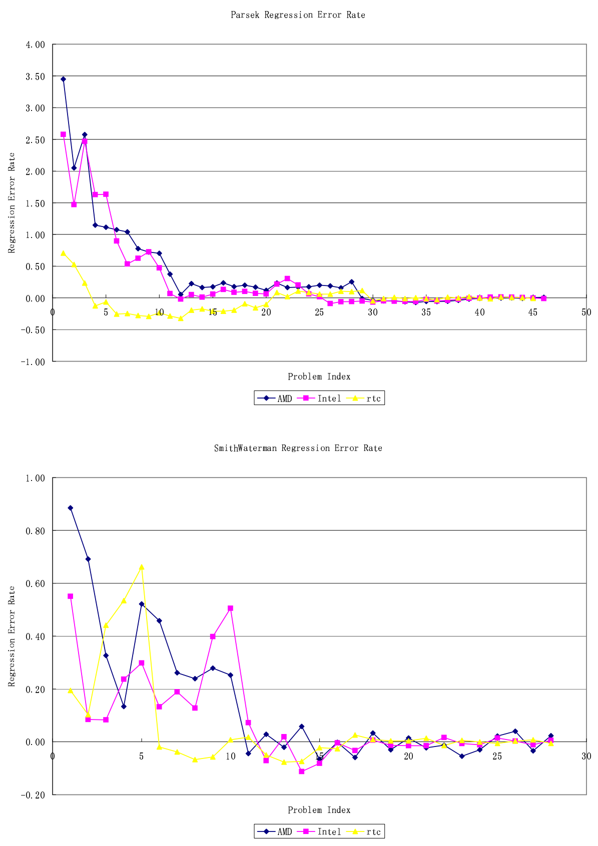

Figure 2. Regression Error Rate of Parsek and SmithWaterman

curves for large size programs. Parsek is a linear algorithm, and all curves on graph (a) in figure 1 match linear pattern quite well. Graph (b) in figure 1 is the execution time of Smithwaterman for different problem sizes on different platform combinations. Smithwaterman is also a linear algorithm, and the the curves match this well.

Graph (a) and (b) in figure 11 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 that can be observed on these figures is that for large problem sizes the regression error rate is very small. This is an encouraging result, suggesting that for large problem sizes, the shape of the execution time curve can be described very well with a simple regression scheme, even with the existence of many complexities in JVM program execution.

Unfortunately, we can also observe some very irregular behavior when problem sizes are small. The reason for this is twofold: the processes inside the JVM like compilation and profiling are most active the beginning of the program execution, and this phase takes up a significant amount of total execution time for small problem sizes. In other words, by the time all the initial analysis and compilation by the JVM is done, the execution is nearing its completion for small problems.

Searching further for the insights into what is happening during the initial phase of the program execution, we profiled the compilation time and number of methods compiled of each application. Table 2 and table 3 contain the data for Parsek and Linpack respectively. Column 2, 3, 4 and 5 represent the time spent on interpreted code execution, compiled code execution, garbage collection and compilation, respectively. The unit in both tables is execution tick, which equals to 20ms.

We can observe on both table 2 and table 3, that time on compilation and time spent on interpretation is rising with problem size, then stabilizing around a certain point. This suggests that the time to find hotspot methods and compile them tends to take a constant amount of time for a large

Particle number	Interpreted	Compiled	GC ticks	Compilation
10	6	N/A	1	80
100	12	0	1	79
1000	23	6	1	91
10000	27	158	2	107
50000	92	1247	12	115
100000	110	2457	32	106
500000	119	12179	150	105
1000000	116	24377	360	99

Table 2. Parsek Profiling Data

Matrix size	Interpreted	Compiled	GC ticks	Compilation
100	11	1	N/A	23
200	8	4	N/A	29
300	14	24	1	42
400	12	87	1	45
500	13	191	5	47
600	15	332	5	28
700	9	549	10	35
800	14	820	8	36
900	13	1186	14	34
1000	12	1651	16	29
1100	10	2186	27	32
1200	15	2839	24	31
1300	8	3599	28	29
1400	16	4488	32	35
1500	47	5526	35	66
1600	12	3372	33	30
1700	17	8015	42	35
1800	16	9490	39	33
1900	14	11198	44	40
2000	12	12089	50	47

Table 3. Linpack Profiling Data

problem. This also explains the more regular execution time behavior for large problems because the effects of the JVM compilation and profiling on total execution time are proportionally smaller for large problems.

Our interpretation of our preliminary results is that the modeling and prediction of the execution time for large problems and for small problems should be handled differently. For large problems, a simple regression analysis may be sufficient to achieve fairly accurate performance prediction, providing that the time complexity model is accurate. For small problem sizes, we have given up hope of accurately modeling and predicting the performance of Java programs – the execution behavior is simply too erratic. We believe however, that the applications that we're primarily targeting – large-scale scientific applications intended for execution on the Grid, mostly fall into the category of "large problems".

The problem of performance modeling and prediction of scientific Java programs now becomes somewhat different and solving it requires solving the following issues:

- How to characterize a given application, with given input size, given the target hardware platform and the target JVM, as "large" or "small"?
- How to determine a set of calibration data for the regression that gives an accurate model, while spending as little time as possible executing the calibration runs?
- How to monitor and predict the garbage collection activity for applications that produce garbage at a higher than a constant rate, and predict the situations when a given problem size for the given application might be "too large" for memory?
- How to determine an accurate execution time model for the application that can be calibrated for particular hardware and virtual machine, using test runs on small problems?
- How to predict the actual execution time of an application given the application itself, input size, target hardware and target virtual machine?

The answer to the first two questions leads us naturally to the concept of Point of Predictability, which we will address in the next section.

The answer to the third question leads us to the concept of Point of Unpredictability, which we will discuss in section 4.

Once we have solved the others, we can solve the last two problems using more traditional methods of code instrumentation and regression modeling, discussed in section 5.

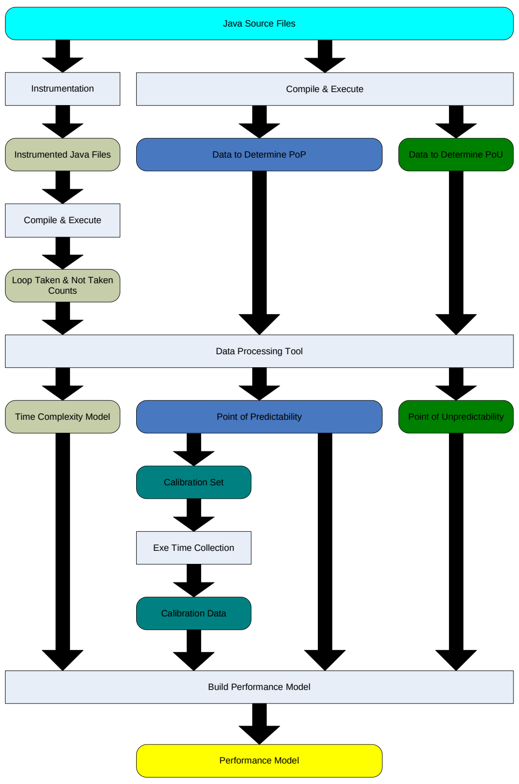

Figure 3. Model Construction Process

Figure 3 shows the work flow graph of the construction process for our performance model. The model construction consists of four parts - time complexity model construction, point of predictability determination, point of unpredictability determination, and performance model construction.

3 Point of Predictability

If we examine the experimental results in the previous section closely, we can observe that for a given application and a given hardware and virtual machine, there is a "turning point" in the problem size, after which the behavior of the application becomes more regular. Moreover, there should exist a problem size such that regression analysis will give consistently good results with a model calibrated using the results from test runs that are larger than that point. Informally, point of predictability should be a point in the problem size space that presents a "turning point": regression analysis gives good results if the model is calibrated using test runs larger than the PoP, and poor results if the model is calibrated using smaller test runs. Of course, this informal definition depends heavily on one's definition of "good" and "poor" in terms of regression analysis.

Even further, we are more interested in the effects of the performance prediction model on load balancing of applications on the Grid, not necessarily in the absolute accuracy of the regression model. In other words, if the regression model is underestimating the performance by 30%, but it's doing so an all the platforms and for all the applications, it should still result in a fairly well load balance schedule.

The realization of this idea is illustrated on graphs (a), (b), (c) and (d) ins figure 4. These figures show the success rate of the load balancing algorithm for different applications when the performance prediction is made using the regression analysis calibrated with five test runs for problem sizes starting with the point on x-axis and increased by 20%.

Let's look at the graph (a) in figure 4 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 problems 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. The "Predicted" curve shows the CPU usage predicted by the load balancing algorithm. Perfect load balance of 100% CPU usage cannot be achieved most of the time even if the performance prediction is 100% accurate – this problem is highly related to the set partitioning problem [5]. The "Actual" curve shows the measured average CPU utilization when the jobs were scheduled using the regression model. The "Frequency based" curve shows, for illustration purposes, the CPU utilization achieved by naïve scheduling algorithm using only processor frequency as the estimate of the performance.

For example, the point for the "Actual" curve on graph (a) in figure 4 for 10000 particles means that when we used the test runs of problem sizes of 10000, 12000, 144000, 17280 and 20736 to calibrate our regression model for performance prediction, the load balancing algorithm produced a schedule resulting in a CPU usage of only 50%, a very poorly balanced execution. On the other hand, using 17280, 20736, 24883, 29859 and 35831 problem sizes to calibrate the model results in a schedule with CPU usage of over 95%, a very well balanced execution.

3.1 Definition

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

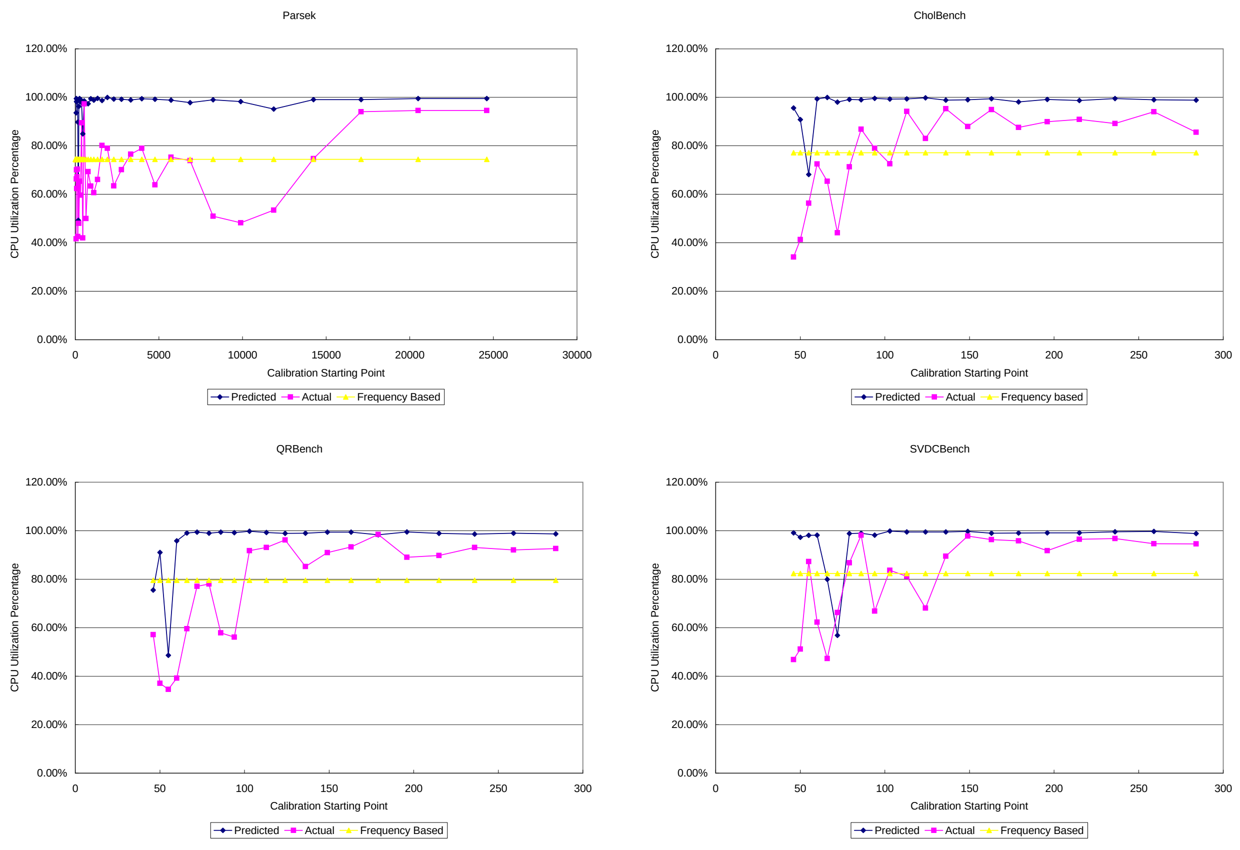

Figure 4. Load Balance

smallest possible CPU utilization is $\frac{1}{n}$ when the platform set contains n platforms.

We used a straightforward greedy algorithm to perform load balancing in the experiments above. The algorithm selects the platform with the shortest job queue at every point and adds a new job to it.

We can observe a common pattern on all the figures in this experiment. Using small problem sizes to calibrate the regression model gives very erratic, and usually very poor results for load balancing. Using large problem sizes for calibration gives consistent results and well balanced schedules. Somewhere in between there should be a point that results in consistent and well balanced schedules, but it isn't any larger than absolutely necessary. We define that point to be the Point of Predictability.

3.2 Determining Point of Predictability

One would be tempted to attempt finding the Point Of Predictability from the data shown on graph (a), (b), (c) and (d) in figure 4. Unfortunately, to collect this data, one would have to run many instances of the problem with the actual problem sizes, making the whole process impractical and defeating the purpose of performance prediction.

Fortunately, we have observed that the number of methods compiled by the JVM is highly correlated with PoP. Once most of the methods of the application have been compiled, the running time of the application ceases to be erratic and starts to follow the time complexity curve.

However, discovering the number of methods currently compiled within a JVM is not a trivial task, since JVM specification does not require the virtual machine to provide that information. The mechanism that does exist ("-XX:+PrintCompilation" flag) unfortunately does not report the methods that are inlined. We modified the JVM code in order to obtain the number of currently compiled and inlined methods from the JVM at run time. We have suggested to Sun to add this information to their JVM requirements, which would make this information available across all platforms.

Using the modified JVM, we were able to perform the experiments shown on graphs (a), (b), (c) and (d) in figure 5. The search for Point of Predictability now transforms into a search for the rightmost plateau on the "Number of methods

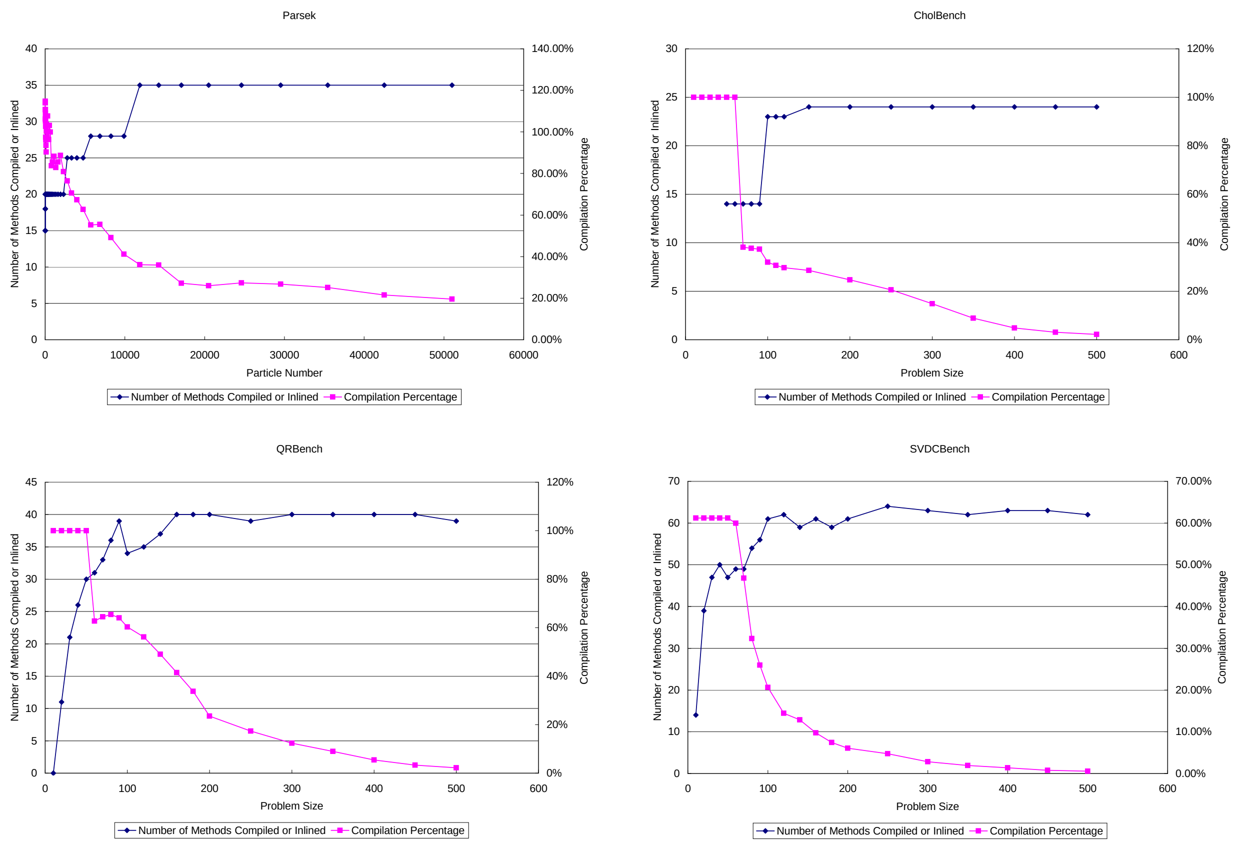

Figure 5. Methods Compiled/Inlined

compiled or inlined" curve.

To achieve this goal, we have taken one more factor into account: the total time spent compiling the code versus the time spent executing. As is to be expected, this factor is fairly large for small problem sizes–the JVM spends most of its time compiling the code. As the problem size increases, the percentage of time spent compiling the code gradually decreases. The "Compilation Percentage" curve on graphs (a), (b), (c) and (d) in figure 5 shows the percentage of the time JVM spends compiling code during the execution of a problem of a given size.

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 3.2. s_0 is the initial problem size from which to start the search. p is the incremental factor for increasing the problem size (linear algorithms require large steps while superlinear require smaller ones). θ is the treshold for the maximum time the JVM should spend compiling that we have empirically determined to be 30%. γ is the standard deviation allowed to consider a collection of runs a plateau, empirically set at 75. n is the number of points to use for plateau detection, empirically set to 5.

4 Point of Unpredictability

We have observed that some applications will enter a "heap-thrashing" mode when trying to execute problems sizes with a footprint that approaches the size of the heap. This is because these particular applications produce garbage at a more than a constant rate, and once the total available heap memory approaches zero, the garbage collector cannot keep up with the garbage production the JVM starts spending most of the time collecting garbage.

This erratic behavior is highly dependant on the size of the heap, and unfortunately makes the execution time unpredictable for certain applications, certain problems sizes and certain heap sizes. Even though the execution time in those cases cannot be actually predicted, identifying those cases without actually running the program of that size would be a useful result. The performance prediction and scheduling module in a Grid execution environment could inform the user that this particular application with this par-

```
inputs:   IN    (initial problem size $s_0$)
                (problem size incremental percentage $p$)
                (compilation percentage threshold $\theta$)
                (standard deviation threshold $\gamma$)
                (size of L $n$)
outputs:  OUT   (point of predictability)
begin
  s = $s_0$
  while $p_c > \theta$
    $p_c$ = $compilation\_pecentage(s)$
    $s = s * (1 + p)$
  for i = 0; i < n; i++
    $L.add(compiled\_method\_num(s), s)$
    $s = s * (1 + p)$
  endfor
  while $STD(L) > \gamma$
    $L.removeFirst()$
    $s = s * (1 + p)$
    $L.add(compiled\_method\_num(s), s)$
  return $L.first().s$
end
```

Figure 6. Algorithm for Determining PoP

ticular problem and heap size cannot be reliably executed on the requested platform, its performance cannot be reliably predicted and it could result in an out of memory error. The user could pursue other means of executing the application – increasing the heap size, rescheduling to a different machine, or searching for a different way to solve the problem.

To handle this problem we introduce another concept: Point of Unpredictability.

Figure 7 shows the garbage collection activities of LUBench for the same problem size under two different maximum heap sizes. We can observe that even though the memory footprint is the same, garbage collection in the case with a 32MB heap happens much more often than in the case of a 128MB heap. The total running time of the application also more than doubles when a 32MB heap is used.

Figure 8 show the total time a virtual machine spends garbage collecting for different problem sizes and for three heap sizes, while figure 9 shows what percentage of the total running time the VM spends collecting garbage.

We can observe that, perhaps surprisingly, the application exhibits a very similar pattern when executing small problems. For example, even though garbage collection takes up to 60% of the execution time when running LUBench with a 300 problem size and a 32MB heap, this figure doesn't change significantly when the heap size is increased. Quadrupling the heap size doesn't improve the performance, the application *still* spends about 60% garbage collecting.

However, once the application memory footprint approaches the size of the heap, we can observe a sharp increase in garbage collection activity. When the application cannot access more heap space, the garbage collector has to collect much more often to free the required memory space for execution. Since garbage collection time is included in the total execution time, this phenomenon will affect the performance model heavily.

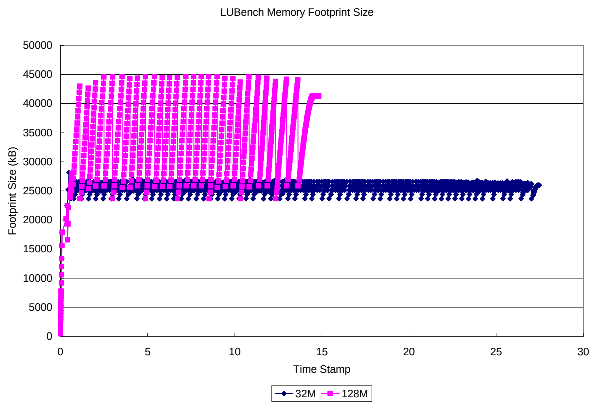

Figure 7. LUBench Memory Footprint Size under Different Max Heap Sizes

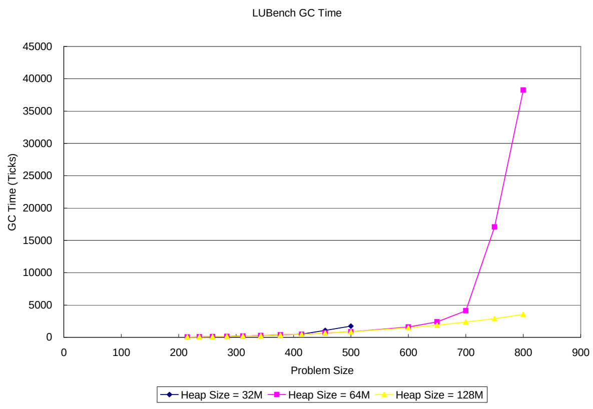

Figure 8. LUBench GC Time under Different Max Heap Sizes

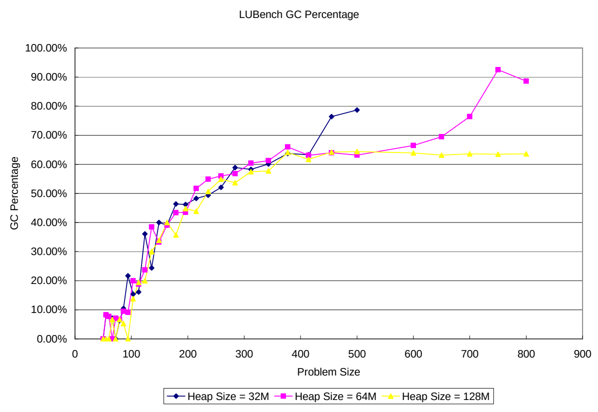

Figure 9. LUBench GC Percentage under Different Max Heap Sizes

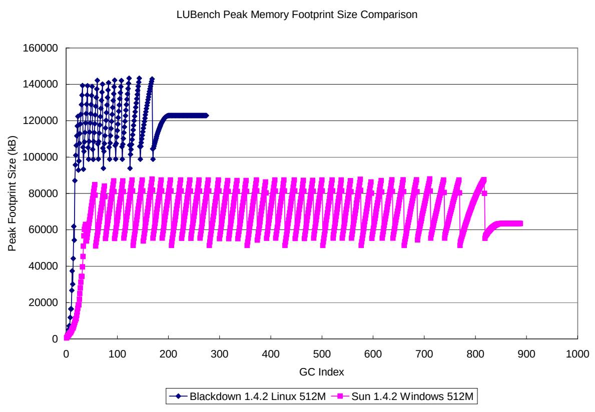

Figure 10. GC Activity Comparison on Different JVM Implementations

4.1 Determining Point of Unpredictability

As with the Point of Predictability, we determine the Point of Unpredictability by executing and analyzing the execution of only a small set of small problems.

First, we need to introduce two terms in order to describe the problem. One is the Base Footprint, which is the minimum memory required by the application for its execution. Another term is the Peak Footprint, defined as the maximum heap memory occupied by the application at any time during the execution, given the problem size and the heap size. The Peak Footprint depends heavily on the garbage collector implementation.

The garbage collector we investigate is the default garbage collector used in Sun's hotspot Java virtual machine. This is a generational garbage collector. A generational garbage collector utilizes the statistical property of the lifespan of heap objects [25]. Statistically, most objects on the heap have a short life span and only a few objects live long. The generational garbage collector split objects into different generations and collect the younger objects more often than older objects. This way, the total time spent on garbage collection can be effectively reduced. Full garbage collections happens much less often. A full garbage collection will scan through all objects on the heap and remove all unreachable objects.

Base Footprint gets reached after a full garbage collection. We monitor the program execution and note the memory size after every full garbage collection. For Peak Footprint, we monitor the memory size at all times. After running a small set of small-size problems, we have enough data to calibrate the regression model for both the Base Footprint and the Peak Footprint as functions of the problem size.

The strategy of determining PoU is based on the capability to predict both Base and Peak Footprint accurately. The strategy is as follows: Select several problem sizes and collect the GC profiling data. Determine the Base and Peak Footprint for each execution. Then perform a regression analysis of both Base and Peak Footprint series to find the best match for functions $f_{Base}(problemsize)$ and $f_{Peak}(problemsize)$. To calculate PoU, solve the equation $f_{Peak}(x) = MaximumHeapSize$. To calculate the problem size when the application will run out of memory (MaxMem), we solve the equation $f_{Base}(x) = MaximumHeapSize$. Between the PoU and the MaxMem, the application will still be able to complete the execution, but its running time will be unpredictably longer than expected, due to the extensive effect of the garbage collection on the running time.

As we stated before, only certain applications will exhibit the "heap thrashing" behavior. In our test suite, the CholBench, QRBenchn, SVDCBench and LUBench are such applications. Others, such as Parsek and Linpack, do not produce garbage at a higher than a constant rate, and as such will never enter a heap thrashing stage. Their Base and Peak Footprints will be the same. Such application will either fit into memory and finish execution with minimal or no garbage collection involved, or it will run out of memory quickly.

PoU is dependent on the implementation of Java virtual machine. For example, figure 10 shows the GC activity for LUBench of the same problem size 678 on two different

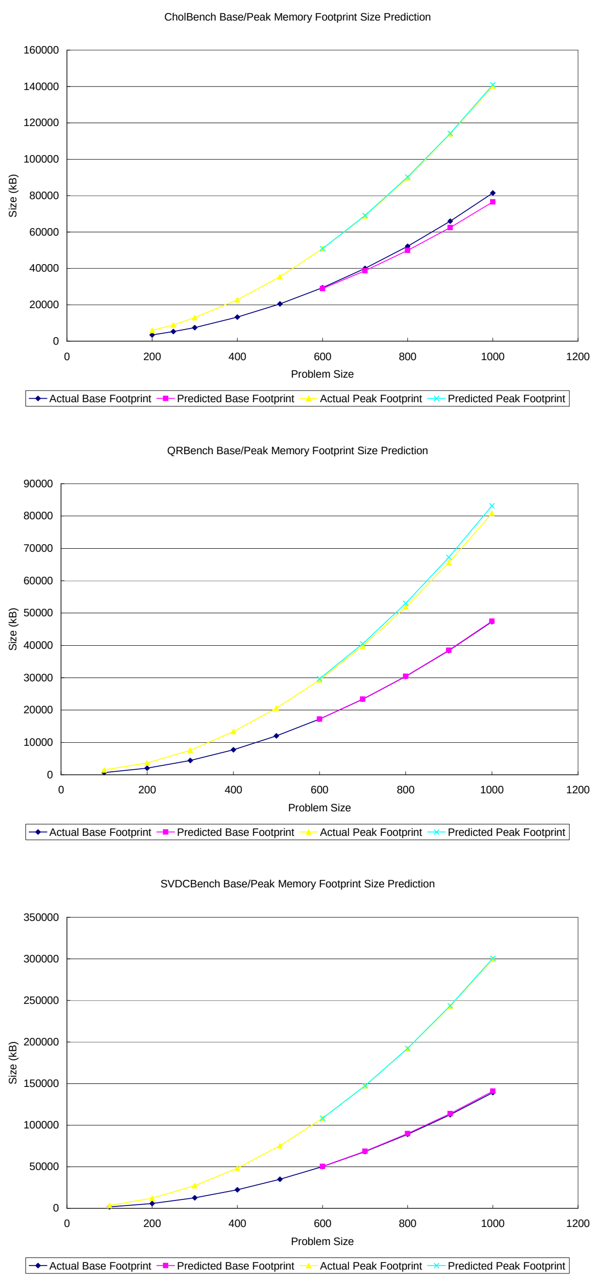

Figure 11. Regression Error Rate of Parsek and SmithWaterman

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, and have a different Peak Footprint. However, on each platform the size of Peak and Base Footprint can be predicted accurately.

Figures **??**, **??** and **??** show the predicted and actual Base and Peak Footprint in function of the problem size for CholBench, QRBench and SVDCBench respectively. The calibration data uses the problem sizes determined by the Point of Predictability, as defined in the previous section. The prediction of both Base and Peak Footprint is quite accurate.

5 Performance Model Construction

Our strategy of model construction is to find the Point of Predictability and Point of Unpredictability (if there exists one) and construct a parameterized model for the problem sizes that in between these two points.

5.1 Collecting Information

For reasons discussed in section 2, we choose to use a profiling-based regression method for building the time complexity model of scientific Java applications.

Straight line code will run straight from the beginning to the end. This kind of code has the simplest time complexity model - O(1). Unfortunately, control flow complicates this issue. Loops and branches will determine the number of times a basic block is executed. We focus on how to gather enough information to describe the behavior of the program.

We achieve this by instrumenting the Java source code. Since code instrumentation can be a prohibitively expensive procedure, we have instrumented only the loops in the program. We insert a counter into each loop to count the times loop has been taken during the execution. We instrument the code, and run a small set of very small problem sizes to collect the loop execution data. We then use the regression analysis to find the complexity function that best describes the behavior of the loop.

Loops are identified by the call site chain. In this context-aware way, we are able to differentiate the same loop called from different paths, which gives a more precise image of how the application is executed.

5.2 Instrumentation and Data Processing

We use a straight-forward source code instrumentation strategy. The instrumentation is performed by pre-pending an instruction before each method call to push the current call site location onto a stack. After each method call, another method call is appended to pop the call site location out of the stack. The stack will contain the information of the call site trace. For each loop in the application, statements to count it's taken times and not taken times are inserted at proper locations as well.

The instrumentation is implemented based on sun's javac compiler and JaMake compiler framework of Rice university [4]. The instrumentation works on the abstract syntax tree built inside of the JaMake compiler.

There is another transformation applied on the program 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 will take these method calls out to be an individual statement. One example is provided below for canonization and instrumentation separately.

Before canonization:

```
z = foo() + goo();
```

After canonization:

```
x = foo();
y = goo();
z = x + y;
```

The identification key for reach loop is created dynamically every time when a loop is encountered. The key is the call site chain string. Statements to update the corresponding counter are inserted at proper positions. After instrumentation, the application will be executed on a set of selected problem sizes to collect the data. Below is one example of the instrumentation.

Before instrumentation:

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

After instrumentation:

```
Counter.callSiteStack.push("key");
x = foo();
Counter.callSiteStack.pop();
```

Following is an example of a loop before and after instrumentation.

Before instrumentation:

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

After instrumentation:

```
Counter.new("key1");
Counter.new("key2");
Counter.increment("key2");
for (i = 0; i<n; i++) {
    Counter.increment("key1");
    {
        x[i] = b[i];
        i++;
    }
}
```

The instrumented code is executed on a set of small problem sizes to collect the counts of how many times the loop has and has not been taken.

We then apply a regression analysis to find the best fit function of each loop. For data-independent applications, we expect that most of their loops will follow a regular behavior.

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

```
Key: SomeKey
Value: [10000, 40000, 90000,
        160000, 250000]
```

To determine the best fit function we perform a search from a pool of predifined complexity functions. For each loop, we attempt to find the best fitting function from the pool, starting with the lowest order ones and continuing to the higher order functions. Once a function fits the data with a predefined treshold, we abort the search – higher order functions will always fit the data better than the lower order ones. Our goal is only to determine the highest order of the polynomial representing the complexity of the program, not the exact complexity function.

A check against all loops returns the highest order. If the highest order is n, the basic performance model function will be $a_1x^n + a_2x^{n-1} + ... + a_nx + a_{n+1}$.

To build the performance model of the application, we collect a set of execution times on different problem sizes. These calibration problem sizes are chosen in the range just beyond Point of Predictability. Then we fit the execution times from the calibration runs onto the time complexity model to construct our performance model.

6 Performance Model Evaluation

To test the effectiveness of our strategy, we have conducted an experiment that schedules 30 instances of the LUBench application onto three different machines while attempting to maximize the load balance.

The tests are done on Andre1, a machine with 2 AMD 1.6GHz Opteron processors, one desktop with AMD Athlon 800MHz and another notebook with Intel Pentium M 1.86GHz processor. The application used for the test is LUBench, which is new for modeling here.

The scheduling process uses the methods described in this paper to determine the Point of Predictability, Point of Unpredictability, the problem sizes to be used for calibration runs, collects the execution times for the calibration runs and uses them to construct a performance model.

It is important to note that LUBench application was not used in any way for constructing the methodology for performance prediction described in this paper nor for collecting the empirical data used for determining PoP, PoU or various thresholds described in previous sections.

Source code instrumentation and complexity model process resulted in a $O(n^3)$ time complexity model for LUBench.

Then we use the strategies presented earlier in this paper to find the Point of Predictability and the Point of Unpredictability. The PoP is 215 for this problem, illustrated on figure 12.

Let problem size be ps, from the data on figure 13 for Sun 1.4.2 JVM on Windows, the regression result for Base Footprint and Peak Footprint are

$$Base = 0.06757 * ps^2 + 9.463 * ps - 861$$

$$Peak = 0.1387 * ps^2 + 23.21 * ps - 1930.$$

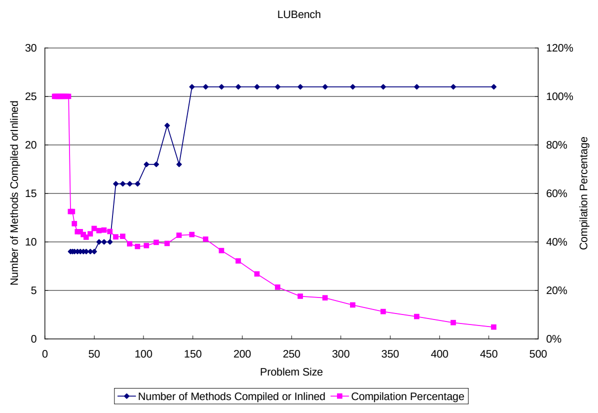

Figure 12. Methods Compiled/Inlined of LUBench

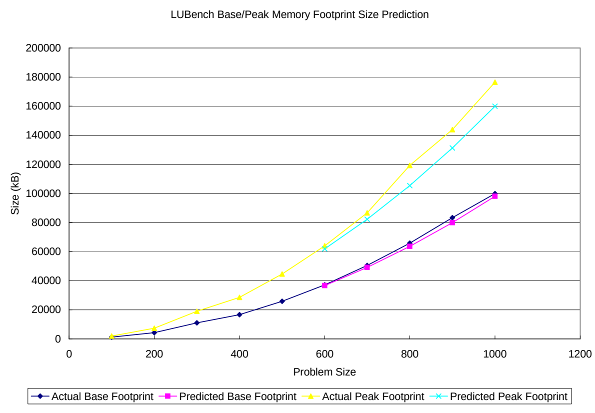

Figure 13. Base/Peak Memory Footprint Size Prediction for LUBench on Sun 1.4.2

PoU is calculated for the case that the maximum heap size is 128MB. PoU is around 800 for Hotspot 1.4.2 windows version. PoU is around 600 for Blackdown 1.4.2 implementation. Figure 16 shows the comparison of execution times between the maximum heap size 128MB and 512MB for Hotspot 1.4.2 for Windows and Blackdown for Linux. The execution times on the 128M curves start diverging from the corresponding curve on 512M, which indicates the PoUs got by prediction are accurate. Figure 16 shows that the PoU predicted for both implementations are accurate.

A set of 30 problem sizes are randomly generated in between of PoP and PoU. The scheduling process then uses our performance prediction to predict the performance of each of those jobs and assign a corresponding weight to them. Then it uses the greedy scheduling algorithm to schedule those jobs on the three platforms described above. The resulting CPU utilization was 0.942, which is a very large improvement over the naïve frequency based strategy that only achieved a CPU utilization of 0.691.

Even though these experiments are preliminary, we can conclude that the techniques described in this paper are highly promising for achieving both practical and effective performance modeling for scientific Java programs, and can be used by an optimizing scheduler to achieve very high load balance.

7 Conclusions

Performance modeling of Java applications is a novel research area. Java execution model introduces serious obstacles to effective and efficient performance modelling.

This paper presents 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, that describe a range in the problem size space where the performance of a Java application can accurately be predicted. We present the techniques for accurately determining these two points by running and analysing only a small set of small-size problems.

We use code instrumentation on very small problem sizes to construct an accurate complexity model for the application. We use Point of Predictability to create a set of calibrating runs that provide the data for a construction of an accurate performance model.

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

Acknowledgments

The authors thank the National Science Foundation for providing partial funds for this work under grant number CCR-0331645. The authors would also like to thank John Mellor-Crummey and Gabriel Marin for their extensive support and help in discussions on performance modelling related issues.

References

[1] G. Ammons, T. Ball, M. Hill, B. Falsafi, S. Huss-Lederman, J. Larus, A. Lebeck, M. Litzkow, S. Mukherjee, S. Reinhardt, M. Talluri, and D. Wood. Wisconsin architectural research tool set. *http://www.cs.wisc.edu/∼larus/warts.html*.

[2] M. Broberg, L. Lundberg, and H. Grahn. VPPB: A visualization and performance prediction tool for multithreaded Solaris programs. pages 770–776.

[3] Z. Budimlić. Owlpack. *http://www.cs.rice.edu/∼zoran/OwlPack/*.

[4] Z. Budimlić and K. Kennedy. JaMake: A Java compiler environment. In *Proceedings of the Third International Conference on Large-Scale Scientific Computations*, Sozopol, Bulgaria, June 2001.

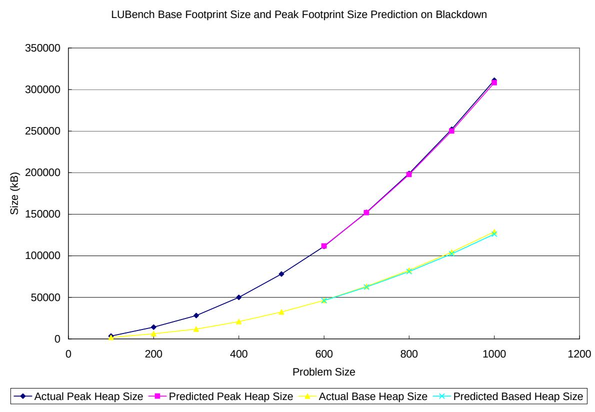

Figure 14. Base/Peak Memory Footprint Size Prediction for LUBench on Blackdown 1.4.2

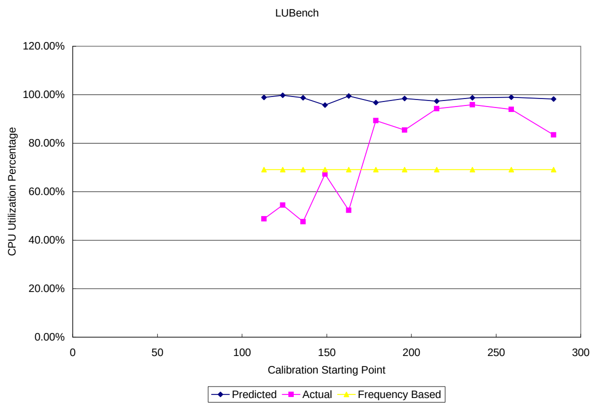

Figure 15. LUBench Load Balance

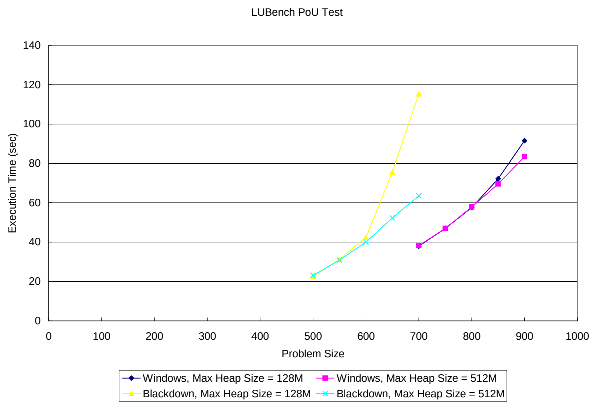

Figure 16. LUBench PoU Test

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. Number 0-262-03141-8. The MIT Press, 1990.

[6] L. Eeckhout, A. Georges, and K. D. Bosschere. How java programs interact with virtual machines at the microarchitectural level. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 169–186, New York, NY, USA, 2003. ACM Press.

[7] I. T. Foster. The anatomy of the grid: Enabling scalable virtual organizations. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 1–4, London, UK, 2001. Springer-Verlag.

[8] H. Gautama and A. J. C. van Gemund. Static performance prediction of data-dependent programs. In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, pages 216–226, New York, NY, USA, 2000. ACM Press.

[9] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. T. Stasko, J. Vetter, and N. Mallavarupu. Falcon: On-line monitoring and steering of large-scale parallel programs. In *Proceedings of the 5th Symposium of the Frontiers of Massively Parallel Computing, McLean, VA,*, pages 422–429, 1995.

[10] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme. A general predictive performance model for wavefront algorithms on clusters of smps. In *ICPP '00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, page 219, Washington, DC, USA, 2000. IEEE Computer Society.

[11] C.-H. A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal, and W.-M. W. Hwu. A study of the cache and branch performance issues with running java on current hardware platforms. In *COMPCON '97: Proceedings of the 42nd IEEE International Computer Conference*, page 211, Washington, DC, USA, 1997. IEEE Computer Society.

[12] M. A. Iverson, F. Ozgüner, and L. Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. *IEEE Trans. Comput.*, 48(12):1374–1379, 1999.

[13] D. Kerbyson, J. Harper, A. Craig, and G. Nudd. Pace: A toolset to investigate and predict performance in parallel systems, 1996.

[14] D. Kerbyson, E. Papaefstathiou, J. Harper, S. Perry, and G. Nudd. Is predictive tracing too late for hpc users, 1998.

[15] A. Mandal, K. Kennedy, C. Koelbel, G. Marin, J. Mellor-Crummey, B. Liu, and L. Johnsson. Scheduling strategies for mapping application workflows onto the grid. In *IEEE International Symposium on High Performance Distributed Computing*, 2005.

[16] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 2–13, New York, NY, USA, 2004. ACM Press.

[17] 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 tool. *IEEE Computer*, 28(11):37–46, 1995.

[18] V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim reference manual. *http://www-ece.rice.edu/∼rsim*.

[19] A. S. Rajan. A study of cache performance in java virtual machines, May 2002.

[20] D. Reed, R. Aydt, T. Madhyastha, R. Noe, K. Shields, and B. Schwartz. An overview of the pablo performance analysis environment, 1992.

[21] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy. The structure and performance of interpreters. *SIGPLAN Not.*, 31(9):150–159, 1996.

[22] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of java workloads: a structured view and opportunities for optimizations. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 194–205, New York, NY, USA, 2001. ACM Press.

[23] A. Snavely, L. Carrington, and N. Wolter. Modeling application performance by convolving machine signatures with application profiles, 2001.

[24] D. Sundaram-Stukel and M. K. Vernon. Predictive analysis of a wavefront application using loggp. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 141–150, New York, NY, USA, 1999. ACM Press.

[25] B. Venners. *Inside the JAVA 2 Virtual Machine*. Number 0-07-135093-4. McGraw-Hill, second edition edition, 1999.

[26] K.-Y. Wang. Precise compile-time performance prediction for superscalar-based computers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 73–84, 1994.