Application-Specific Benchmarking

A thesis presented

by

Xiaolan Zhang

to

The Division of Engineering and Applied Sciences

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

May, 2001

Abstract

This thesis introduces a novel approach to performance evaluation, called application-specific benchmarking, and presents techniques for designing and constructing meaningful benchmarks.

A traditional benchmark usually includes a fixed set of programs that are run on different systems to produce a single figure of merit, which is then used to rank system performance. This approach often overlooks the relevance between the benchmark programs and the real applications they are supposed to represent. When the behaviors of the benchmark programs do not match those of the intended application, the benchmark scores are uninformative, and sometimes can be misleading. Furthermore, with the rapid pace of application development, it is impractical to create a new standard benchmark whenever a new “killer” application emerges.

The application-specific benchmarking approach incorporates characteristics of the application of interest into the benchmarking process, yielding performance metrics that reflect the expected behavior of a particular application across a range of different platforms. It also allows benchmarks to evolve with applications and consequently the benchmarks are always up-to-date.

This thesis applies the application-specific benchmarking methodology to a variety of domains covering Java Virtual Machines, garbage collection, and operating system. The result is a collection of benchmark suites that comprise the HBench framework, including HBench:Java, HBench:JGC, and HBench:OS.

This thesis demonstrates HBench's superiority in predicting application performance over a more conventional benchmarking approach. It is shown that HBench:Java is able to correctly predict the rank of running times of three commercial applications on a variety of Java Virtual Machine implementations. In the realm of garbage collection, the predicted garbage collection times for a stop-the-world, mark-sweep garbage collector closely match the actual times. In the domain of operating system, it is demonstrated that HBench:OS can be used to analyze performance bottlenecks and to predict performance gains resulting from a common optimization.

Acknowledgments

During my years at Harvard I am fortunate enough to have worked with a group of brilliant scholars. It is their enthusiasm towards research that inspires me; it is their encouragement that drives me to achieve the best I can. I owe them a great deal of gratitude.

Among them is my advisor Margo Seltzer, to whom I am eternally grateful for her guidance and support, especially during the last two years when I had to work remotely due to the two-body problem. Her visionary view of systems research has proved invaluable to my thesis research. Her wonderful personality together with her high standards for research makes her an ideal advisor. To me, she is more than an advisor – she is also a role model.

Brad Chen advised me for the first three years of my study. His creative thinking has had tremendous impact on me. What I learned from him has been very helpful in my research. Mike Smith provided sound advice in my earlier projects. Mike is also one of the nicest people I know. This is evidenced by the ping-pong table he donated to the division, which dramatically improved the quality of life for many graduate students. Professor Michael Rabin showed me the beauty and elegance of randomized algorithms. His pursuit for simplicity has profound influence on my approach to problems. Jim Waldo has the rare combination of both academic strength and a strong sense of business. I am grateful for the countless help and advice I received from him.

During my summer internships I have come to know a few great researchers who helped shape my thesis research. Bill Weihl's group at Digital's System Research Center taught me many things about architecture. Lance Berc helped me find the Java applications examined in this thesis. In the summer I spent with Steve Heller's group at Sun Microsystems Lab East, I learned about garbage collection (GC). That experience greatly assisted my research on benchmarking garbage collectors. Dave Detlefs and Ole Agesen know more about GC than anybody else I can think of. Their insights and guidance have helped me develop a deeper understanding about GC.

I also wish to express gratitude to my fellow students and colleagues who have made this journey more enjoyable. The VINO group members deserve my thanks for their assistance in many aspects. I have enjoyed many long interesting conversations with Keith Smith; I could not thank Kostas Magoutis enough for many rides to the airport and for numerous lunch discussions on research ideas; Most of my research was conducted on machines "borrowed" from Dan Ellard; Dave Sullivan and Dave Holland have also helped in various ways. Thanks to the HUBE group, most notably Cliff Young, Zheng Wang, and Glen Holloway, for their support in my earlier projects. My daily life would be less pleasant without the company of my other fellow students, Rebecca Hwa, Racqell Hill, Xianfeng Gu, Chris Small, Yaz Endo, Chris Stein and Dave Krinsky. Many thanks to Qingxia Tong for her friendship and for hosting me during the last few weeks of my stay in Boston. I am also grateful to Chris Lindig for our weekly ping-pong game, which has been my only physical exercise for the recent years. Finally, Elizabeth Pennell and Mollie Goldbarg also helped in proofreading my papers.

This thesis is dedicated to my family, my husband Zhibo Zhang, my parents, Ruhuai Zhang and Huizhen He. Without their continuous love and support, none of this would have been possible.

Table of Contents

Chapter 1
Introduction

"In the computer industry, there are three kinds of lies: lies, damn lies, and benchmarks."

- Unknown

A benchmark is defined as a set of programs (or micro-programs) that are run on different systems to give a measure of their performance. Results of standard benchmarks, therefore, reflect only the performance of the set of programs on the system being measured. In reality, there is often a mismatch between programs included in standard benchmarks and real applications. Consequently, standard benchmark results can be uninformative, and sometimes even misleading. We identify two sources of this gap:

First, the set of programs included in a standard benchmark is fixed. However, applications used by end-users can vary widely. Frequently, the programs included in standard benchmarks are not representative of real applications that end-users care about. The problem can be made worse by vendor over-optimization. As benchmark rating becomes a more important factor in the buying decision, vendors design their systems around popular benchmarks. Some vendors even over-optimize their systems for certain standard benchmark to gain a few percentages points in the benchmark score, often at the expense of the performance of real applications. Such incidents have been reported in graphics benchmarks, where graphics card vendors employ an "enhancement" technique to improve their results in the WinBench [62] Graphics WinMark benchmark, while

severely hampering the performance of other devices in realistic situations [26]. The fact that the set of programs included in standard benchmarks is generally fixed facilitates this over-optimization practice, making it even harder to judge the true performance of a system in the presence of a real application. It is therefore not surprising that many computer professionals take a cynical view of benchmarks.

Secondly, as the trend of exponential growth in hardware speed and in software complexity continues, benchmark development has been lagging behind due to both economic and technical reasons. The cost of developing benchmarks has been increasing steadily due to the increasing complexity of applications. As a result benchmark development cycles have lengthened. On the other hand, fierce competitions force companies to roll out new versions of applications at an accelerated rate. The implied result is that standard benchmarks often lose out in the technology race and "represent yesterday's workloads", as observed by Dixit in a lecture given at the 2001 High Performance Computing Architecture conference [14]. The SPEC WEB99 [55] benchmark is a good example. SPEC WEB99 is a standard benchmark for measuring web server performance and is endorsed by major web-server vendors. The first version of SPEC WEB, SPEC WEB96, was introduced in 1996, three years after the world-wide-web took off, and contained only static files in the workloads. The second version, SPEC WEB99, was introduced in 1999 and included dynamically generated contents. However it was again two years after popular usage of dynamic web pages. How long SPEC WEB99 can stay current (or whether it is still current today) remains to be seen. This imbalance between application development and benchmark development inevitably

System		Standard Benchmark Results		Actual Results (email/sec)
		SPEC CPU95	PostMark (transaction/sec)	
A		6.22	16	2.2
B		6.23	6	4.6
	Winner	none/both	A	B

Table 1-1. Standard Benchmark Results vs. Actual Application Performance on Two Similarly Priced Systems. Both benchmarks were run in single-user mode. For the actual results, the same version of the *sendmail* mail server was used on both machines. A fast client machine sends email as quickly as the server machine can receive, via a dedicated 100Mb/s Ethernet switch.

leads to the situation that a gap forms between available benchmarks and state-of-the-art applications.

1.1 How Benchmark Results Can be Misleading – An Example

The gap between standard benchmark results and real application performance can be demonstrated with the following simple example. Imagine a hypothetical scenario where you were a system administrator, and you were asked to buy a new mail server machine for your department. Suppose that you narrowed down the choice to two, system A and system B with similar price tags. Which one would you select as the mail server? A common approach is to run a standard generic benchmark on the two systems and pick the one with the higher rating. An alternative approach is to use a more specific benchmark that measures certain aspects of the system that are potentially important to the application's performance. Table 1-1 shows the results for SPEC CPU95 [52], a

widely used benchmark for measuring CPU performance[1], and those for PostMark [30], a file system benchmark designed specifically for modern applications such as mail and news servers. The results of SPEC CPU95 suggest that the two systems are comparable. From the results of PostMark we can conclude that system A is a better choice. However, measurements of the actual mail server performance show that system B can handle more than twice the email traffic than system A can. This example demonstrates that standard benchmarks often do not represent the behavior of real applications. Consequently, the benchmark scores offer little indication of real applications' performance.

1.2 Thesis Contributions

The goal of this thesis is to establish that traditional ways of performance evaluation is flawed, and that new approaches that can better predict application performance need to be developed. This thesis introduces one such approach called *application-specific benchmarking*, and presents techniques for designing and constructing benchmarks that reflect performance of real applications.

The basic idea of application-specific benchmarking is to separate characterization of an application from that of the underlying platform and combine the two characterizations to form a prediction of the application's performance. As the application of interest is incorporated into the "benchmarking" process, the resulting performance metrics reflects the expected behavior of the application on the given platform.

[1] At the time this research started, there was no standard mail server benchmark. In January of 2001, SPEC released its first mail server benchmark, SPEC MAIL2001.

The main contributions of this thesis are as follows. This thesis applies the application-specific benchmarking methodology to a variety of domains covering Java Virtual Machines, garbage collection, and operating system. Two benchmark suites are created as a result of this, HBench:Java and HBench:JGC. This thesis also demonstrates HBench's predictive power from three different perspectives. It is shown that HBench:Java is able to correctly predict the rank of running times of three commercial applications on a variety of Java Virtual Machine implementations. In the realm of garbage collection, the predicted garbage collection times for a stop-the-world, mark-sweep garbage collector closely match the actual times. In the domain of operating system, it is demonstrated that HBench:OS can be used to analyze performance bottlenecks and to predict performance gains resulting from a common optimization.

1.3 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 starts with a brief overview of benchmarks in general and then describes some domain-specific benchmarks in the areas of Java Virtual Machine, garbage collection, and operating system.

Chapter 3 describes the HBench approach and the methodology developed within the HBench framework for measuring systems with different degrees of complexity. One approach, the vector-based methodology, forms the foundation for the three benchmark suites presented in this thesis, namely, HBench:Java, HBench:JGC, and HBench:OS.

Chapter 4 applies the vector-based methodology to the domain of Java Virtual Machines. It describes the challenges faced during the design process of HBench:Java and the solutions to these issues. It also describes the microbenchmark suite included in

HBench:Java and the profiler implementation on Sun Microsystems JDK1.2.2 platform. Results on a variety of JVMs demonstrate HBench:Java's superiority over traditional benchmarking approaches in predicting real application performance and its ability to pinpoint performance problems.

Chapter 5 applies the vector-based methodology to evaluating garbage collector performance. Although the discussion is restricted to the context of garbage collection in the Java Virtual Machine, techniques presented in this chapter apply to garbage collection in general. Experimental results on the Sun JDK1.2.2 JVM implementation on three machine configurations show that the predicted garbage collection times track the actual elapsed times closely.

Chapter 6 applies the methodology to evaluating operating system performance in the context of the Apache web server. It is shown that HBench:OS can be used to analyze performance bottlenecks and to predict performance gains resulting from a common optimization.

Chapter 7 summarizes the research findings and lessons learned and presents some new research directions.

Chapter 2
Background and Related Work

This chapter first gives an overview of computer benchmarks in general and then discusses traditional benchmarks in three specific areas, namely, Java Virtual Machines, garbage collection, and operating systems, for which we have implemented benchmark suites using the application-specific benchmarking methodology and evaluated their effectiveness.

2.1 Computer Benchmarks in General

2.1.1 Types of Benchmarks

There are two categories of benchmarks in terms of benchmark size and complexity, namely, *microbenchmarks* and *macrobenchmarks*. Microbenchmarks measure performance of primitive operations supported by an underlying platform, whether hardware or software. Sometimes microbenchmarks also include short sequences of code (kernels) that solve small and well-defined problems. Typically the mean (typically geometric mean) of the individual times (or scores as a function of the time) is reported. Examples of primitive operations include the time it takes to fetch a data item from cache/memory, or the time it takes to draw a line on a graphical terminal. Microbenchmarks reveal a great deal of information about the fundamental costs of a system and are useful in comparing low-level operations of different systems, but it is difficult to relate them to actual application performance in a quantitative way.

Macrobenchmarks consist of one or more medium-scale to large-scale programs that are usually derived from real applications. Macrobenchmarks usually come with

input data that are supposedly representative of typical situations. When the benchmark is run, the input data are fed to the programs and the total running time is collected. As macrobenchmarks are typically derived from real applications, they place a significant amount of stress on the underlying system like real applications do. Therefore, the benchmark result can be a good indication of the system's performance, if the normal load on the system is the same as the programs included in the benchmark.

2.1.2 Examples of Current Standard Benchmarks

Several standard institutes are in charge of the process of defining and distributing standard benchmarks. Some of the well-known standard entities are: SPEC [56] (System Performance Evaluation Corporation) and TPC [57] (Transaction Processing Performance Council).

SPEC defines a wide variety of standard benchmarks, ranging from high-performance computing to network file servers. Among those, the SPEC CPU benchmark suite [52] is probably the most widely used benchmark in computer literature. SPEC CPU is a macrobenchmark that is further divided into two macrobenchmarks, the integer suite and the floating-point suite. The integer benchmark suite consists of popular UNIX desktop applications such as `gzip` [21], a GNU compression program, `gcc` [17], a GNU C language compiler, and `perl` [60], a scripting language created by Larry Wall.

TPC is another standard entity that specializes in benchmarks for transaction processing and database systems. TPC has produced widely used benchmarks such as TPC-A, TPC-B, and their successors TPC-C and TPC-D. TPC-C is an OLTP (OnLine Transaction Processing) benchmark that emulates an interactive order processing system.

TPC-D is decision support benchmark that contains a set of complex business-oriented queries. The most prominent feature of TPC benchmarks is that they report price/performance ratio in the benchmark results, a useful metric that helps customers balance additional performance and cost of the system. The ratio is reported in the form of dollars per *tpm* (transaction per minute). Current versions of TPC benchmarks require significant effort to set up and run. The cost for a TPC-C run, for example, can reach a half million dollars [19].

2.2 Non-Traditional Approaches to Benchmarking

Many researchers have realized the problem with fixed-workload benchmarks [41] and have proposed different solutions to this problem [20][51].

In recent work [20], Gustafson et al. present a novel approach of measuring machine performance that, instead of fixing the task size of the benchmark programs, fixes the time and measures the amount of work that gets done during the fixed time period. The benchmark program (called *HINT* for Hierarchical INTegration) tries to use interval subdivision to find rational bounds on the integral of the function $(1-x)/(1+x)$ where $x \in [0,1]$. The amount of work being done in each step is measured by the improvement on quality of the bound (i.e., the distance between upper and lower bounds). Because there is no upper limit to the tightness of the bound (subject to the available precision), HINT is able to scale with the power of the computer being measured. A fundamental difference between HINT and HBench is that the HINT program consists of a fixed mix of instructions (e.g., the ratio of computation operations to memory operations is about 1). HINT is a good performance indicator for applications

that have a similar instruction mix, but not necessarily for applications with different instruction mixes. HBench, on the other hand, is a general framework that adapts to a specific application.

Dujmovic et al. [15] propose a systematic approach to constructing benchmark programs. They employ a *block-frame-kernel (BFK)* model that recursively constructs benchmark programs using basic building blocks such as control structures and *kernels*, which are either simple statements or a short sequence of code that implements a specific function. A benchmark program consists of a number of *blocks* containing one or more *frames*, each of which in turn may include kernels or blocks. Parameters of the models include block *breadth*, which denotes the number of frames included in the block, and block *depth*, which denotes the level of nesting. A benchmark program is characterized by the probability distributions of these two parameters, the probability distribution of control structures, and the kernel code. The authors suggest that by carefully selecting kernels to be included in the benchmark, they can construct benchmark programs that model any kind of workload.

This approach could conceivably reduce benchmarking development time greatly, resulting in much shorter benchmark production cycle. However, this approach represents only an incremental step in the science of benchmarking. Even though benchmark programs can be produced more quickly, they still need to be run on all possible target configurations to determine the performance. If the application behavior changes, a new benchmark must be constructed and run again on each target configuration. HBench, on the other hand, tries to predict application performance. As we shall see in Chapter 3, with HBench, once system performance data is collected on a target machine, there is no

need to run the application or benchmark on the target machine to obtain a prediction on the particular machine's performance with regard to the application, even if the application of interest changes over time.

2.3 Performance Evaluation of Java Virtual Machines

Introduced by Sun Microsystems Inc. in the late 90's, Java quickly became the popular programming language for the Internet age. The Java concept contains three parts, the Java programming language, the Java *bytecode*, and the *Java Virtual Machine* (JVM) [2]. Programs written in the Java programming language are compiled into hardware-independent Java bytecodes, which run on Java Virtual Machines. Java bytecode is a stack-based language that contains a small set of instructions. Java Virtual Machine, whose implementation is hardware-dependent, can understand and execute programs in bytecode format. The hardware-independency of Java bytecode has proved to be a big win – once a Java application is compiled into bytecode, it can be executed anywhere a JVM is present. This releases programmers from the mundane task of porting code to different platforms and allows rapid application development.

There has been a proliferation of Java Virtual Machine implementations and Java benchmarks since the introduction of the Java technology. By and large the benchmarks can be classified into three groups, microbenchmark, macrobenchmark, and a combination of the two.

CaffeineMark [9] is a typical example of microbenchmarks. It measures a set of JVM primitive operations such as method invocation, string manipulation procedures, arithmetic and graphics operations. CaffeineMark was once a popular benchmark for

JVMs embedded in web browsers, when the most usage of Java came from Java *applet*, a small program in bytecode format that can be downloaded and executed in the local browser environment. CaffeineMark runs as an applet and requires minimum configuration to run.

There are many macrobenchmarks for Java, among which SPEC JVM98 is probably the most popular [53]. The SPEC JVM98 suite includes a set of programs similar to those found in the SPEC CPU suite, such as a compression program, and a Java compiler, all written in Java. VolanoMark [59] from Volano LLC is another popular Java macrobenchmark. VolanoMark is designed to address performance concerns of the company's Java-based VolanoChat™ server. As a server benchmark, VolanoMark focuses on a JVM's ability to handle long-lasting network connections and threads.

The JavaGrande benchmark suite [8][38] consists of both microbenchmarks and macrobenchmarks. Designed to compare the ability of different Java Virtual Machines to run large-scale scientific applications, the JavaGrande benchmark suite contains three sections. The first section consists of microbenchmarks that measure low-level operations of a Java Virtual Machine. Examples include arithmetic operations such as addition, multiplication and division, mathematical functions such as *sin*() and *cos*(), and exception handling, etc. The second section consists of *kernels*, each of which contains a type of computation likely to appear in large scientific programs. Examples include Fourier (Fourier coefficients computation) and Crypt (IDEA encryption and decryption), etc. The final section includes realistic applications, such as a financial simulation based on Monte Carlo techniques.

This hybrid approach of combining micro and macro benchmarking provides the ability to reason about performance disparities between Java Virtual Machines, and is particularly useful in pinpointing performance anomalies in immature Java Virtual Machine implementations. For example, based on the results of the JavaGrande suite, the authors pointed out that the IBM HPJ compiler performs worse than the other JVMs in the Fourier kernel benchmark, mainly due to an inferior performance in mathematical functions [8].

This type of reasoning, although proved useful in the domain of scientific applications, has its limitations. In order to associate the performance of the micro-benchmarks with that of the application, the user is assumed to have intimate knowledge about the application, which is seldom true in practice, especially for large and complicated applications. Even if the user knows the application well, when the application's performance depends on more than a handful of primitive operations, it's hard to explain what causes the differences and to predict the application's performance on a new JVM.

2.4 Garbage Collector Performance Evaluation

Garbage collection is a well-known technique for automatic memory management. Automatic memory management frees programmers from the burden of explicitly maintaining reachability information of the data structures and remembering to free those data structures when they become unreachable. Garbage collection is commonly used in functional languages such as Lisp, Scheme and ML. As garbage collection technology advances, the speed of garbage collection has improved

significantly over the years and is almost comparable to the speed of explicit memory management [67]. Since automatic memory management provides additional advantages of improved programming productivity and less error-prone code, garbage collection is employed in many modern object-oriented languages such as Java and Smalltalk. Even system programming languages such as C++ have started to use garbage collectors.

Many researchers have studied the performance of dynamic memory management [13][67][49]. This literature provides a good foundation for understanding the inherent cost of dynamic storage allocation. Our approach differs in the goals we try to achieve. We emphasize predictability — the ability to predict application performance on different GC implementations without running the application on the target implementations. In contrast, past research has focused on comparing the cost of memory management by running a set of popular applications on target memory management implementations.

Knuth presents a comprehensive analysis and comparison of the time complexity of several dynamic storage management algorithms [31]. This systematic approach to benchmarking memory management algorithms offers insight into the efficiency of these algorithms and helps explain the performance differences. However, the analysis assumes certain statistical properties for both memory allocation and liberation patterns and only applies when the system reaches equilibrium.

Cohen et al. compare performance of four compacting algorithms using analytical models [12]. The analytical models are parameterized by the amount of work to be done, such as the number of cells (objects), number of pointers (links) and related information, and the time to perform the basic operations common to all compactors, such as the time

to test a conditional expression. Their goal is similar to ours in that they also try to estimate GC execution times “without resorting to empirical tests”. The main difference lies in the level of abstraction used for the primitive (elementary) operations. Their primitive operations are low-level machine instructions, whereas we conglomerate all machine instructions performed on an object into a single per-object operation (e.g., per-live-object overhead). Because their primitives are at such a low-level, their models are more elaborate and require intimate knowledge of the algorithms (i.e., the complete source code). Furthermore, as computer architectures become more advanced, machine-level optimizations and the memory cache hierarchy could introduce significant side effects such that the analytical model will no longer be applicable. In our case, the cost of primitives is measured explicitly by microbenchmarks and therefore includes these side effects.

2.5 Operating System Benchmarking

Microbenchmarks that measure the costs of basic kernel primitives, such as thread creation and inter-process communication latencies, are often used for characterizing performance of operating systems, especially microkernels [4] [22]. Ousterhout presented a comparison of operating system performance using a test suite mostly comprised of kernel microbenchmarks [42]. The benchmark results revealed differences between RISC and CISC architectures in terms of OS performance, and provided insights into why speed-ups in hardware do not translate to the same speed-up in operating systems. Lmbench, created by Larry McVoy [39], represents more recent development in kernel microbenchmarking. Lmbench contains a richer set of OS and hardware primitives and is more portable, making it a valuable tool for analyzing system performance. As in the case

of Java microbenchmarks, although kernel microbenchmarks reveal important performance details about the primitive operations, they alone are not sufficient to determine how fast a given application would run on the operating system in question.

Another school of OS benchmarks concentrates on a particular subsystem such as file system and network system. The Postmark benchmark [30], for example, is a file system benchmark designed to model the file access patterns of mail servers. Subsystem benchmarks are similar to kernel microbenchmarks, but typically allow one to specify probability (weight) for each operation so that the mix of operations more closely approximates the real workloads. However, the results usually provide only a partial picture on the overall performance.

When comparing operating system performance, researchers often measure end-to-end performance of kernel-intensive applications. Benchmarks of this sort are by definition, macrobenchmarks. Web servers are a popular choice, although more often researchers just use *ad-hoc* programs. It is extremely difficult to develop macrobenchmarks for real-world applications, as can be seen from the lack of standard news server and mail server benchmarks until fairly recently [47] [54].

2.6 Performance Prediction Using Queueing Models

Queueing models have been used extensively in performance modeling and analysis. They are based on the client-server scenario, where clients arrive at a certain rate, possibly wait in the queue for a certain amount of time, and are then served by the server for a fixed amount of time. Some systems, such as transaction processing systems, network channels, and time-sharing computing servers, naturally fit into this client-server

scenario. One advantage of a queueing model is its simplicity. The number of parameters in a typical queueing model is quite small, and they can usually be estimated with sufficient accuracy from known performance characteristics of the computer hardware. For systems whose performance can be described with a simple queueing network model, i.e., they satisfy certain assumptions, the queueing models could produce quite accurate results – past research indicates that the expected accuracy of queueing network models is between 10% and 30% for response time [32]. This level of accuracy is sufficient for most cases.

However, not all systems can be described with a queueing model. A garbage collector, for example, is not a client-server type of application by its nature. In that sense, we believe that the HBench approach is complimentary to the queueing model approach to performance evaluation.

2.7 Conclusions

There is no lack of benchmarks in any field of computer science[2]. However, traditional benchmarks fail to address the issue of relevance to real applications. Real-world applications are so diverse that it is difficult, if not impossible, to find a set of workloads that are representative of all the important applications, even within a sub-field. If the behavior of the benchmark's workloads does not match that of the intended application, then the benchmark might give misleading information regarding which platform is the best for the application of interest. This is precisely the problem that the

[2] The benchmarks discussed in this chapter are by no means a comprehensive list of benchmarks used in the industry and the research community. Readers are recommended to read [18] and [19] for a more detailed coverage about this topic.

application-specific benchmarking approach tries to tackle. I hope that the techniques for application-specific benchmarking presented in this thesis will help computer scientists conduct more meaningful performance comparisons and encourage the industry to participate in searching for better benchmarking methodologies.

Chapter 3
Application-Specific Benchmarking: the HBench Approach

This chapter presents the design of HBench, our approach to application-specific benchmarking, and describes the vector-based methodology that forms the basis of the three benchmark suites included in this thesis.

3.1 The HBench Approach

HBench is designed with the belief that systems should be measured in the context of applications in which end-users are interested. HBench achieves this by incorporating the application of interest into the benchmarking process. The process consists of three stages. First, we characterize the system using well-defined and application-independent metrics based on standard specifications that summarize the fundamental performance of the system. For example, hard drive vendors describe the performance of a hard disk in standard terms such as RPM (Rotations Per Minute) and seek latency of the disk head. Next, we characterize applications with a system-independent model that captures the amount of demand that the application places on a system in terms of the standard metrics for system characterizations. Finally we combine the two characterizations using an analyzer to predict the running time of the application on the given system. Figure 3-1 depicts the schema of the HBench process. Because HBench incorporates application characteristics into the benchmarking process, HBench can provide performance metrics that reflect the expected behavior of a particular application on a particular platform.

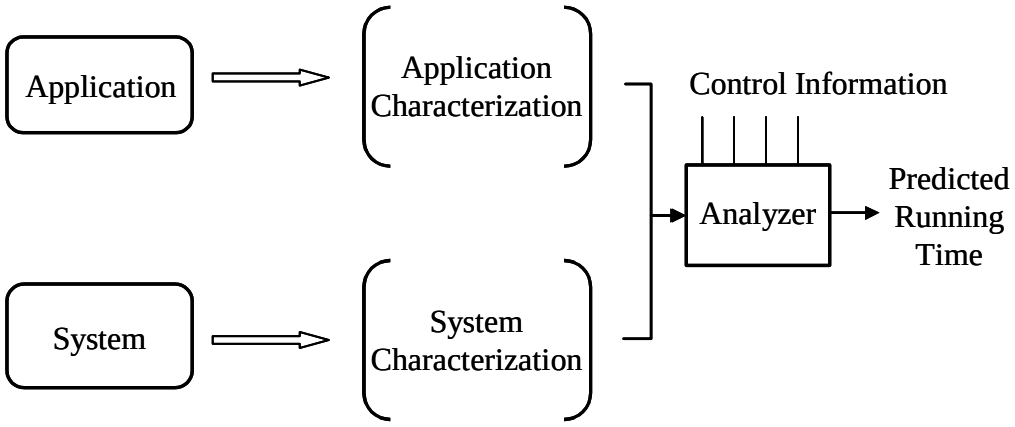


Figure 3-1. Schematic View of the HBench Process

Different systems have varying degrees of complexity, requiring different techniques for application and system characterization and for combining the two characterizations. HBench includes three methodologies, namely, the vector-based methodology, the trace-based methodology, and the hybrid methodology. This thesis primarily focuses on the vector-based methodology. The other two are included in this discussion for the purpose of completeness. The following subsections describe them in detail.

3.1.1 The Vector-Based Methodology

In general, a system's performance can be determined by the performance of each individual primitive operation that it supports. A given application's performance can then be determined by how much it utilizes the primitive operations of the underlying system. As the name "vector-based" indicates, we use a vector $V_s = [v_1, v_2, ..., v_n]$, to represent the performance characteristics of an underlying system, with each entry v_i representing the performance of a primitive operation of the system. We call this vector V_s a *system vector*, and it is obtained by running a set of microbenchmarks.

HBench uses an *application vector*, $V_A = [u_1, u_2, ..., u_n]$ to represent the load an application places on the underlying system. Each element u_i represents the number of times that the corresponding ith primitive operation was performed. The application vector is typically obtained through profiling. The dot product of the two vectors produces the predicted running time of the application on a given system.

The vector-based approach has been applied to a number of platforms including Java Virtual Machines [65], garbage collectors [66] and operating systems and has demonstrated excellent predictive power. In the domain of operating systems, the primitives are system calls and popular functions in standard C libraries.

The complicated structure of JVMs presents some challenges for identifying the primitives. Currently HBench focuses on two components of the JVM – standard Java class APIs and the garbage collector. Chapter 4 describes in detail how HBench applies the vector-based methodology to JVMs using Java class method APIs as primitive operations. Chapter 5 presents techniques for application-independent characterization of garbage collectors, for characterizing applications' memory behavior independent of the underlying garbage collector, and for combining the two characterizations to form a figure of merit.

Primitives of a garbage collector are operations performed on objects, such as allocation, marking and copying. Sometimes, the primitive's performance is dependent on the size of the object. For example, the time of a copy operation depends on the size of the object. The system vector for such operation would contain the cost of the operation for every object size, as depicted in the following:

$[cost_{op,size=1}, cost_{op,size=2}, \ldots, cost_{op,size=n}]$.

The corresponding application vector would look like:

$[count_{size=1}, count_{size=2}, \ldots, count_{size=n}]$.

As the size of objects can get very large, a more convenient way to represent the two vectors would be to use distribution functions when they are available or can be devised. Let $C_{op}(s)$ denote the cost for operation *op* as a function of object size *s*, and let $N(s)$ be the object-size distribution function. Then the total cost can be simply denoted by

$$\sum_{s} C_{op}(s) \cdot N(s),$$

or more precisely,

$$\int_{0}^{\infty} C_{op}(s) \cdot N(s) ds .$$

The basic strategy behind HBench has been to use the simplest model possible without sacrificing accuracy. To that end, we use a simple linear model, until we find that it is no longer able to provide the predictive and explanatory power we seek. In some cases, rather than going to a more complex model, we retain the simplicity of a linear model by adding multiple data points for a single primitive.

3.1.1.1 Advantages

The vector-based methodology offers several desirable features. First, it allows meaningful comparison to be made between systems' performance. The system and application vectors provide an effective way to study and explain performance

differences between different system implementations. Secondly, the vectors reveal performance bottlenecks that are useful in guiding performance optimizations in a way that will benefit the application of interest. System implementers can improve primitive operations that are significant for the application. At the same time, application programmers can optimize the application by reducing the number of calls to expensive primitive operations. Finally, The vector-based methodology allows one to predict the performance of the application on a given system without actually running the application on it, as long as the system vector is available. One might also answer "what if" questions such as "What if this primitive takes half as much time?" by modifying the appropriate system and application vector entries. This feature is particularly useful for capacity planning and for designing next generation products.

3.1.1.2 Limitations

The vector-based approach has its limitations, however. The underlying assumption of this methodology is that each primitive takes a fixed amount of time to execute, independent of the context in which it is invoked. The methodology also requires that for a given application, the number of times each primitive is invoked is fixed, or can be calculated deterministically. These restrictions limit the situations to which this approach can be directly applied. For example, in the case of virtual memory, the number of page faults of a given application depends on page reference patterns of other processes as well. Consequently, the number of page faults for a given application cannot be decided ahead of time, even if we know the page replacement algorithm. For experiments conducted in this thesis, we assume that the application's working set fits in main memory. Distributed systems are another case that the vector-based approach does

not apply directly, since message latencies in such a system can vary widely. However, even with these limitations, in practice we find the model applicable to sufficient number of cases that it can be used to predict the performance of real-world applications, as demonstrated in Chapters 4, 5 and 6.

3.1.2 The Trace-Based Methodology

The trace-based methodology is designed for applications whose performance relies on the sequence of input data (e.g., web requests). Web servers, for example, exhibit drastically different access patterns [36]. In the trace-based approach, the trace (or log) of the input request stream, instead of the application, is characterized with a stochastic model. The model can then generate request sequences that approximate the actual request stream, to be used as a benchmark. Furthermore, one can vary the load by adjusting parameters of the model, such as the number of users and the size distribution of the file set, allowing one to conduct capacity planning experiments. Manley et al. present techniques on characterizing both the web sites and the user access patterns from logs of web servers, and demonstrated how these characterizations can then be used to generate load that closely matches the original load and to scale the load to reflect growth patterns [37]. Wollman extends these techniques to measure the performance of proxy servers [63]. In addition to characterizations of the file and user access patterns, latencies between the proxy server and actual web sites were derived from logs of the proxy server. Wollman demonstrated that HBench:Proxy can generate heavy bursty traffic that cannot be achieved by other proxy benchmarks.

3.1.3 The Hybrid Methodology

The hybrid approach is a combination of the previous two approaches, and is designed for systems whose primitive operation interferes with each other. In such systems, depending on the sequence of the operations, the same operation might take different amount of time to execute. For example, in the case of the file system, a file read operation might require different amount of time to complete, depending on whether the data is in the file cache or on the disk.

Currently, the hybrid approach is used in HBench:FS, one of the benchmark suites in the HBench collection, to measure file system performance [50]. The system vector measures cache-specific file operations, e.g., it measures both a cache-hit read and a cache-miss read. Then the application trace is fed to a cache simulator, which generates an application vector whose elements have specific meaning in terms of the system vector, e.g., the number of cache-hit reads and the number of cache-miss reads. Again, the dot product of the two vectors produces the estimated time spent in the file system.

The idea of using the cache simulator to iron out the interdependencies of file system primitive operations might be adapted to the virtual memory subsystem, so that the number of page faults can be estimated and accounted for in the total running-time prediction.

3.2 Related Approaches

The transition from traditional benchmarks to a vector-based approach resembles the transition from MIPS ratings to the decoupled approach proposed by John Hennessy and David Patterson in analyzing performance of RISC (Reduced Instruction Set

Computer) architectures [23]. A MIPS rating is obtained by running an arbitrary program chosen by the vendor. Therefore, it is not informative in judging how fast a particular application will run on the given machine, since the instruction mix used by the application is likely to differ from that of the program used by the vendor to derive the MIPS rating.

John Hennessy and David Patterson revolutionized the performance reporting methods for computer architectures in the mid 80's. Instead of using a single metric such as MIPS, their method characterizes a computer's performance with CPIs (Cycle Per Instruction) for every instruction supported by the computer. This data, when combined with the number of times each instruction is executed by a particular application, yields the total number of cycles the computer spends executing the application of interest, which in turn yields the total running time in seconds.

The vector-based approach of HBench is also similar to the abstract machine model [45], where the underlying system is viewed as an abstract Fortran machine, and each program is decomposed into a collection of Fortran abstract operations called *AbOp*s. The machine characterizer obtains a machine performance vector, whereas the program analyzer produces an application vector. The linear combination of the two vectors gives the predicted running time. This approach requires extensive compiler support for obtaining the accurate number of AbOps and is limited to programming languages with extremely regular syntax. It is also highly sensitive to compiler optimization and hardware architecture [46]. As hardware becomes more sophisticated, the accuracy achievable with this technique tends to decrease.

3.3 Conclusions

In conclusion, HBench is a realistic and constructive approach to benchmarking. When applied appropriately, it can provide both meaningful comparisons and valuable information to system and application developers for future improvement on their products.

The effectiveness of HBench lies in its ability to predict application's performance. We identify three levels of predictive power: prediction on relative performance, prediction on ratios of running times, and prediction on actual times. The initial goal is to have HBench predict the correct order of relative performance, while keeping the vectors small. The vectors can then be improved to achieve more accurate prediction.

Chapter 4
HBench:Java: An Application-Specific Benchmark for JVMs

In this chapter, we demonstrate how we applied the vector-based methodology of HBench to evaluating Java Virtual Machine performance. The first task is to identify primitive operations for Java Virtual Machines. This topic is treated in Section 4.1. Section 4.2 describes the prototype implementation of HBench:Java, the microbenchmark suite that measures primitive costs. Section 4.3 presents experimental results. Section 4.4 discusses some unresolved issues and Section 4.5 concludes.

4.1 Identifying JVM Primitive Operations

4.1.1 JVM Overview

A JVM is a complicated piece of software [58]. Figure 4-1 shows a schematic view of a JVM implementation. Much of a JVM's functionality is supported via the system classes (also called built-in classes or bootstrap classes). Sun Microsystems publishes the specification of these abstractions, which is supported by any JVM implementation that conforms to the Java standard.

Like many modern programming languages, memory management in Java is automatic. A JVM includes a memory management system (also called the garbage collector) that automatically frees objects when they are no longer referenced by the application.

The execution engine is responsible for interpreting Java bytecode, resolving and loading classes, and interfacing with native methods (methods that comprise of native

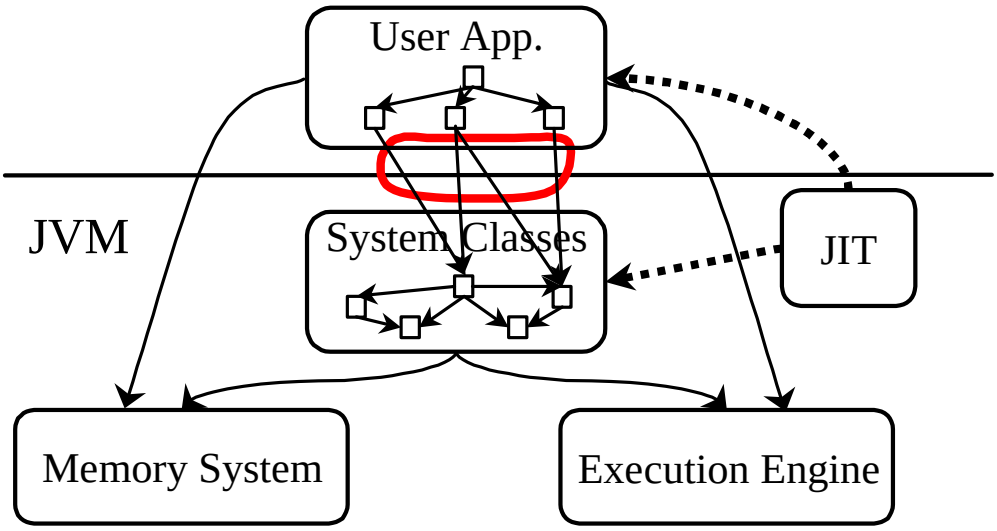


Figure 4-1. Schematic View of a JVM.

machine code instead of Java bytecode). It also performs tasks similar to operating systems, such as thread scheduling and context switches, exception handling, and synchronization.

The JVM implementation is further complicated by the just in time (JIT) compiler component, which compiles Java bytecode into native machine code on the fly. In some newer versions of JVM implementations, the JIT also performs various types of dynamic code optimizations to improve the performance of the application.

4.1.2 First Attempt

In order to create a system vector for a JVM, we need to decompose this complexity into a set of primitive operations. At first glance, the JVM assembly instruction (Java bytecode) seems to be a perfect candidate. The Java Virtual Machine instruction set includes about 200 instructions [35], which is sufficiently small for a

```
// empty loop
for (int i = 0; i < numIterations; i++) {
      ;
}

// loop containing integer addition
for (int i = 0; i < numIterations; i++) {
      sum += i;
}
```

Figure 4-2(a). Java Code Sequences

```
//empty loop
loop_start:
      inc    ecx                 ;; i++
      cmp    ecx, [esi+04h]      ;; i<numIterations
      jnge   loop_start

// loop containing integer addition
loop_start:
      add    edi,ecx             ;; sum += i
      inc    ecx                 ;; i++
      cmp    ecx, [esi+04h]      ;; i<numIterations
      jnge   loop_start
```

Figure 4-2(b). Corresponding Native Code Sequences

complete microbenchmarking to be possible. Bytecode is also universal - all flavors of JVM implementations support it.

This approach, however, proved inadequate primarily due to the presence of the JIT. Once bytecodes are compiled into native machine code, optimizations at the hardware level such as out-of-order execution, parallel issue and cache effects can lead to a running time that is significantly different from the sum of the execution times of the individual instructions executed alone.

For example, Figure 4-2(a) shows two Java code sequences: an empty loop and a loop containing an integer addition operation. The corresponding native code produced

by the JIT is shown in Figure 4-2(b). On a Pentium III processor, both loop iterations take 2 cycles to execute, due to parallel instruction issues. This leads one to conclude that the addition operation is free, which is clearly not true.

4.1.3 A Higher Level Approach

A higher level of abstraction that is immune or less sensitive to hardware optimization is therefore needed. We identified the following four types of high-level components of a JVM system vector:

- system classes, with method invocations to the system classes being primitive operations;

- memory management, where primitive operations could include object allocation, live-object identification (marking), live-object relocation (for copying garbage collectors) and dead-object reclamation (see chapter 5 for more details);

- execution engine, where primitive operations include bytecode interpretation, exception handling, context switching, synchronization operations, etc.;

- JIT, which can be measured by two metrics: overhead and quality of code generated. JIT overhead can be approximated as a function of bytecode size, in which case the primitive operation is the time it takes to JIT one bytecode instruction. The product of this per-bytecode overhead and the number of JITted bytecodes yields the overall overhead. Note that the number of JITted bytecodes cannot be directly obtained from the application, as it is JVM dependent. Rather, it is obtained by applying a JVM dependant function *J* to the base application vector *M*, and *S*, where each entry in *M* and

S represent each method's invocation count and bytecode size, respectively. For example, if a JVM compiles a method the first time it is invoked, then

$$J(N,S) = \sum_i s_i$$

where s_i is the ith element of *S*. The quality of JITted-code is harder to quantify, and is a subject of ongoing research.

The system classes component provides a convenient abstraction layer and is a good starting point for our prototype implementation of HBench:Java. This chapter focuses on the system classes component only, as highlighted by the circle in Figure 4-1. Therefore, HBench:Java at its current stage is intended for applications that are system-classes bound. Our experience shows that applications tend to spend a significant amount of time in system classes. Therefore we believe that this simplistic system vector, albeit crude, can be indicative of application performance. Our results demonstrate that HBench:Java already provides better predictive power than existing benchmarks.

4.2 HBench:Java Implementation

The implementation of HBench:Java consists of two independent parts: a profiler that traces an application's interactions with the JVM to produce an application vector and a set of microbenchmarks that measures the performance of the JVM to produce a system vector. The following two sub-sections describe these parts in more detail.

4.2.1 Profiler

The profiler is based on JDK's Java Virtual Machine Profiling Interface (JVMPI) [29]. Once attached to the JVM, a profiler can intercept events in the JVM such as

method invocation and object creation. The Java SDK1.2.2 kit from Sun comes with a default profiling agent called *hprof* that provides extensive profiling functionality [33]. We use this default profiler to obtain statistics of method invocations from which we derive an application vector. As a first step, our application vector (and accordingly our system vector) only contains method invocations to JVM system classes.

A drawback of JVMPI is that it does not provide callbacks to retrieve arguments of method calls. To remedy this problem, we implemented a second profiler that is able to record method arguments; it is based on JDK's Java Virtual Machine Debugger Interface (JVMDI) [28]. Since JVMDI can only be enabled with JIT turned off (for the classic version of JDK), we keep both profilers for obvious performance reasons, with the first profiler responsible for extensive profiling and the second profiler responsible for the much simpler task of call tracing for a subset of primitives.

4.2.2 Microbenchmarks

The current set of microbenchmarks consists of approximately thirty methods including frequently invoked methods and methods that take a relatively long time to complete, based on traces from sample applications. Even though these methods represent only a tiny portion of the entire Java core API, we found them quite effective in predicting application performance, as shown later in Section 4.3.

The microbenchmark suite is implemented using an abstract Benchmark class. To add a microbenchmark to the suite, one implements a class that extends the Benchmark class. Specifically, this means implementing the `runTrial()` abstract method. A utility program facilitates this process by automatically generating the corresponding source

Java program from a template file and a file that specifies key information about the particular microbenchmark.

Typically, the `runTrial()` method invokes the method to be measured in a loop for some number of iterations. A nice feature of our microbenchmarks is that the number of iterations is not fixed, but rather dynamically determined based on the timer resolution of the `System.currentTimeMillis()` function of the specific JVM. A microbenchmark is run long enough that the total running time is at least n times the timer resolution (to allow for accurate measurement), and less than $2n$ times the timer resolution (so that the benchmark doesn't run for an unnecessarily long time). For the experiments reported in this thesis, we used a value of 10 for n.

For methods whose running time also depends on parameters, such as the `BufferedReader.read()` method that reads an array of bytes from an input stream, we measure the per-byte reading cost and the corresponding entry in the application vector includes the total number of bytes instead of the number of times the `read()` method is called. The current prototype implementation supports this simple case of linear dependency on a single argument, and it is found sufficient for the sample applications tested. For more complicated argument types, the system vector entry would consist of a list of $(n+1)$-tuples, $(t, a_1, a_2, \ldots, a_n)$, where a_i is the value of the ith argument, and t is the time it takes to invoke the method with the given arguments. We then measure several data points in this n-dimension space and extrapolate the running time based on the actual parameters included in the corresponding application vector entry.

Method Name	Method Signature	Time(us)
java.lang.Character.toString	()Ljava/lang/String;	2.498
java.lang.String.charAt	(I)C	0.092
java.io.BufferedReader.read	([CII)I	6.897
java.lang.Class.forName	(Ljava/lang/String;)Ljava/lang/Class;	5309.944
java.net.Socket.<init>	(Ljava/net/InetAddress;I)V	2171.552

Figure 4-3. Sample Microbenchmark Results.

Figure 4-3 shows some sample microbenchmark results for JDK1.2.2 (Windows NT). The time for the `read()` method of `BufferedReader` is the per-byte read cost, and the `Class.forName()` method loads an empty class.

4.2.3 JVM Support for Profiling and Microbenchmarking

For some primitive operations such as class loading, the first-time invocation cost is the true cost and subsequent invocations just return a cached value. As a result we cannot simply measure the cost by repeatedly calling the method with the same arguments in a loop and dividing the total time by the number of iterations. In the case of class loading, it means we need to load a different class for every iteration. With the timer resolution of current JVM implementations, to achieve reasonable accuracy, the number of iterations required is on the order of hundreds and increases as processor speed increases. We could automatically create these dummy classes before starting the loop. However, not only does this approach not scale well, creating a large number of class files also perturbs the results since the number of classes within a directory is usually not that large. A better solution is to have the JVM provide a high-resolution timer API. This approach has the added advantage of reduced benchmark running time (recall that the number of loop iterations is inversely proportional to the timer resolution). Most modern

CPUs provide cycle counters that are accessible in user mode, and many popular operating systems such as Solaris and Windows NT already provide high-resolution timer APIs.

One of the difficulties of microbenchmarking is that sometimes a good JIT will recognize the microbenchmark code as dead code and optimize it out. We have to insert code to fool the JIT into believing that the variables used in the microbenchmark loop are still live after the loop, and subsequently not optimized out of the loop. However, there is a limit as to how much this workaround can do. A better solution would be for the JIT to include command-line options that allow users to specify optimization levels, similar to those present in C/C++ compilers.

Advanced JIT techniques such as the adaptive compilation used in HotSpot [25] pose some difficulties measuring JIT overhead, which cannot be overcome without help from JVM implementers. An adaptive compiler compiles methods based on their usage. Methods might be interpreted initially. As time progresses, some are compiled into native code with a lightweight compiler (with little optimization). Frequently executed methods might be re-compiled with a more powerful backend compiler that performs extensive optimization. The problem lies in how to model the JVM dependent function *J* which, given the number of method invocations and method bytecode sizes, yields the number of bytecodes compiled/optimized. We think the following enhancement to JVM would be useful:

- A JVMPI event should be generated at the beginning and end of the compilation of a method, so that we can model and evaluate *J*.

JVM	CPU	Memory (MB)	Operating System	JVM Version	Vendor
JDK1.2.2_NT_PRO	Pentium Pro 200MHz	128	Windows NT 4.0	1.2.2 Classic	Sun Microsystems
SDK3.2_NT_PRO				5.00.3167	Microsoft
JDK1.2.2_NT_II	Pentium II 266MHz	64		1.2.2 Classic	Sun Microsystems
SDK3.2_NT_II				5.00.3167	Microsoft
JDK1.2.2_SunOS_ Classic	Ultra SPARC IIi 333 MHz	128	Solaris 7	1.2.2 Classic	Sun Microsystems
JDK1.2.1_SunOS_ Prod				1.2.1_O3 Production	Sun Microsystems

Table 4-1. Java Virtual Machines Tested.

- To measure the per-bytecode compiler/optimize overhead, the `java.lang.Compiler` class should be augmented with APIs for compiling and optimizing methods.

4.3 Experimental Results

4.3.1 Experimental Setup

The experiments were run on a variety of Java Virtual Machines. Table 4-1 shows the list of JVMs tested and their configurations.

Three non-trivial Java applications (Table 4-2) were used to evaluate HBench:Java. First, the applications were run with profiling turned on, and we derived application vectors from the collected profiles. For Mercator, which is a web crawling application, the proxy server and the web crawler were run on two different machines connected with a 100Mb Ethernet switch, isolated from the outside network. The

Application	Description	Input Data
WebL	A scripting language designed specifically for processing documents retrieved from the web [61].	A WebL script that counts the number of images contained in a sample html file.
Cloudscape	A Java- and SQL-based ORDBMS (object-relational database management system). The embedded version is used, i.e., the database is running in the same JVM as the user program [11].	The JBMSTours sample application included in the Cloudscape distribution kit. Only the BuildATour program, which simulates the task of booking flights and hotels, is used.
Mercator	A multi-threaded web crawler [40].	The synthetic proxy provided by the Mercator kit that generates web documents on the fly instead of retrieving them from the Internet.

Table 4-2. Java Applications Used in the Experiments.

machine that hosted the proxy server was at least as fast as the machine that hosted the client, to insure that the proxy server was not the bottleneck. Next the HBench:Java microbenchmarks were run on the JVMs listed in Table 4-1, which produced their system vectors. The dot products of the system and application vectors gave the estimated running time for each application on each JVM, which was then compared with the actual running time to evaluate the effectiveness of HBench:Java.

Since the initial goal is to correctly predict the ratios of execution times of the applications on different JVM platforms, normalized speeds were used in reporting experimental results. This also allows one to compare HBench:Java with conventional benchmarking approaches such as SPECJVM98 that report results in the form of ratios.

4.3.2 Results

Figure 4-4 shows the results for the scripting language WebL. In this experiment, three primitive operations account for the majority of the running time, shown in Table 4-3. Also shown in Table 4-3 is their measured performance on the five Java Virtual Machines tested. The corresponding application vector is (80, 121, 32768). It is interesting to note that the SPECJVM98 score of JDK1.2.2 on the PentiumPro NT machine is higher than that on the SPARC workstation. However, WebL runs close to three times as fast on the SPARC workstation. HBench:Java's system vector reveals the problem. Class loading is twice as fast for the SPARC workstation JDK, and the `BufferedReader.read()` method executes almost 35 times faster. It turns out that for some reason, the NT JDK1.2.2's JIT didn't compile the method `sun.io.ByteToCharSingleByte.convert()`, an expensive method called many times by `java.io.BufferedReader.read()`. The differences result in superior performance on the SPARC workstation. Besides explaining performance differences, the predicted ratios of execution speeds are within a small margin of the real execution speed ratios.

Figure 4-5 shows the results for Cloudscape, a database management system. The result for the Sun JDK1.2.2 classic version on the SPARC workstation was missing because Cloudscape wasn't able to run on it. Similarly to what we observed for the WebL results, not only does HBench:Java correctly predict the order of the running speed on the different JVM platforms, the predicted ratios of the execution speeds closely match the actual ratios. On the other hand, SPECJVM98 does not predict the order correctly, and its predicted speed ratios are off by a large margin in most cases. Also similar to the case of WebL, Cloudscape spends a large amount of time in class loading.

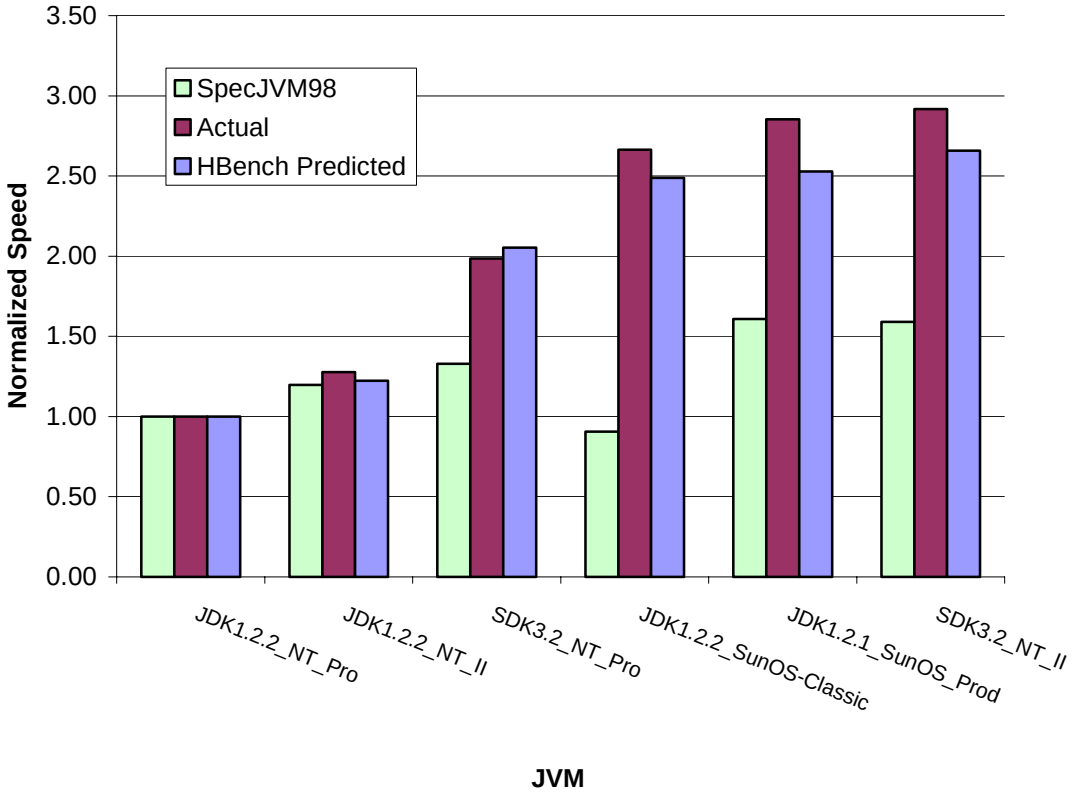


Figure 4-4. Normalized Running Speeds for WebL. The speeds are normalized against the reference JVM, JDK1.2.2_NT_PRO, i.e., the normalized speed of SDK3.2_NT_PRO is the execution time on JDK1.2.2_NT_PRO divided by the execution time on SDK3.2_NT_PRO. This graph shows that HBench:Java is able to correctly predict the order of the running speed of WebL on the different JVM platforms.

JVM	Time (µs)		
	Class.forName()	ClassLoader.loadClass()	BufferedReader.read()
JDK1.2.2_NT_PRO	5309.944	4564.824	6.897
SDK3.2_NT_PRO	3011.411	2710.269	0.317
JDK1.2.2_NT_II	4155.065	3961.282	5.108
SDK3.2_NT_II	2281.390	2053.251	0.244
JDK1.2.2_SunOS_Classic	2264.093	2037.331	0.195
JDK1.2.1_SunOS_Prod	2487.306	2145.458	0.139

Table 4-3. Important Primitive Operations for WebL.

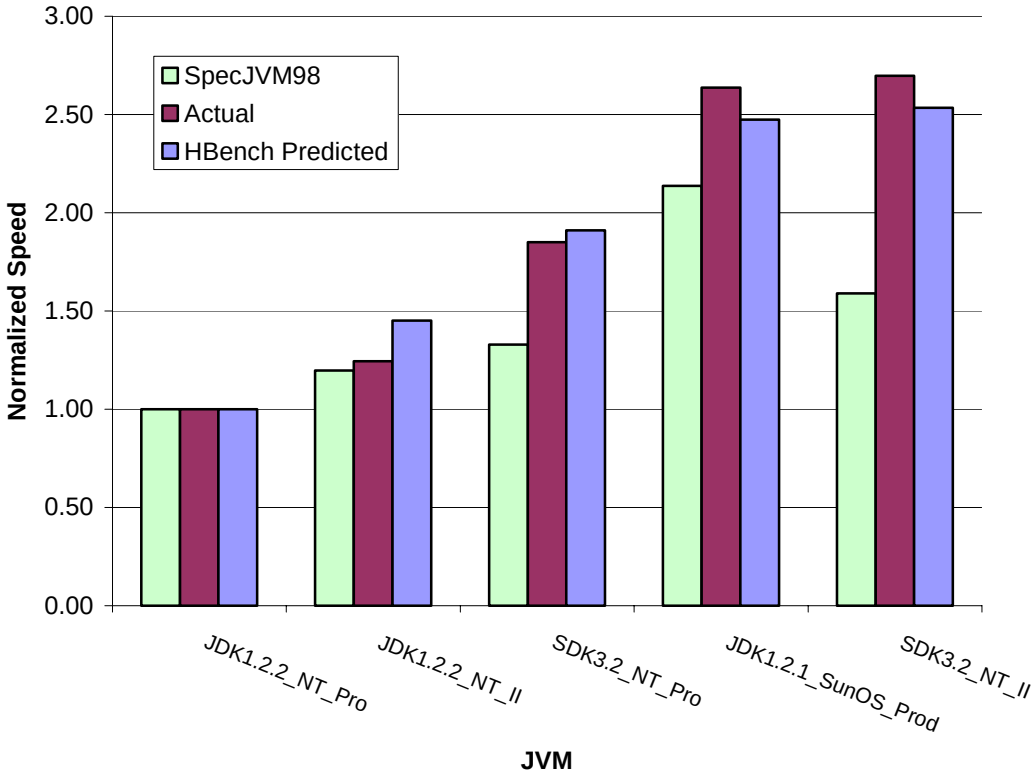


Figure 4-5. Normalized Running Speeds for Cloudscape. The speeds are normalized against the reference JVM, JDK1.2.2_NT_PRO. This graph shows that HBench:Java is able to correctly predict the order of the running speed of Cloudscape on the different JVM platforms.

Figure 4-6 shows the results for Mercator, the web crawler. Only results for a limited number of JVMs were collected due to the difficulty of setting up the machines in an isolated network. The results, however, are quite encouraging. Even though HBench:Java predicted the order for JDK1.2.2_NT_Pro and SDK3.2_NT_Pro incorrectly, the predicted ratio still matches the actual ratio quite closely. As a matter of fact, the actual ratio is so close to one, it is difficult to tell which one is faster. SPECJVM98 again predicted the wrong order for Sun JDK1.2.2. In this case, two primitive operations, the constructor of `java.net.Socket` and `java.net.SocketInputStream.read()`, account for the majority of the running time.

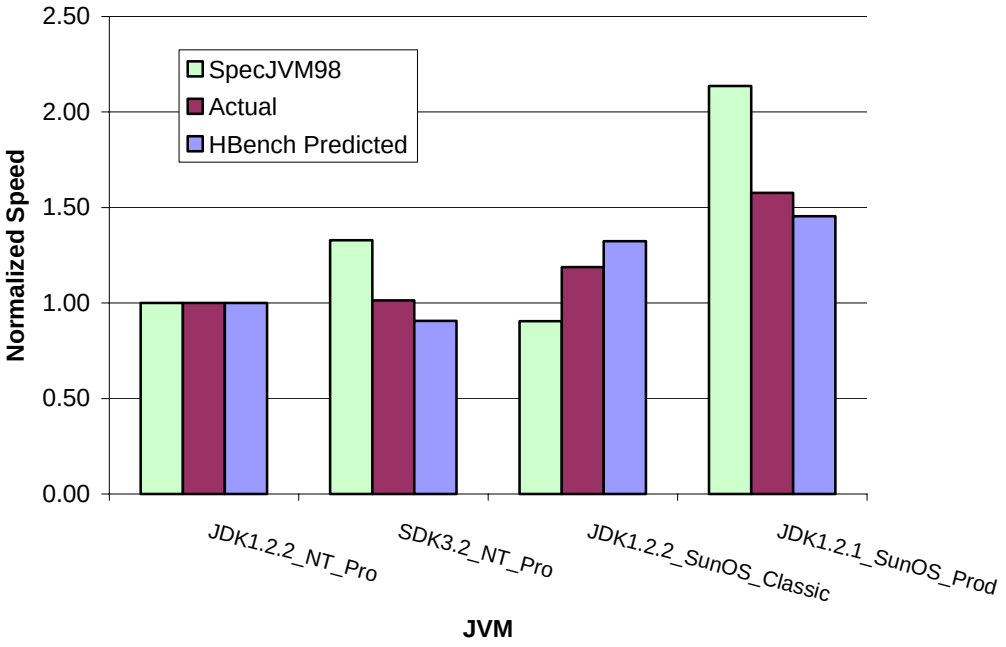


Figure 4-6. Normalized Running Speeds for Mercator. The speeds are normalized against the reference JVM, JDK1.2.2_NT_PRO. This graph shows that HBench:Java is able to correctly predict the order of the running speed of Mercator on the different JVM platforms.

JVM	Time (µs)	
	Socket.<init>()	SocketInputStream.read()
JDK1.2.2_NT_PRO	2171.552	0.210
SDK3.2_NT_PRO	2575.459	0.214
JDK1.2.2_SunOS_Classic	826.780	0.262
JDK1.2.1_SunOS_Prod	660.711	0.254

Table 4-4. Important Primitive Operations for Mercator.

Table 4-4 lists the cost of these two primitives for the four Java Virtual Machines tested. The per-byte socket read time is quite similar for the four JVMs. The socket initialization time, which includes the cost of creating a TCP connection, varies a lot among the four JVMs. The corresponding application vector entry is (19525, 147550208).

Program	System Time (%)	User Time (%)
_201_compress	2.6	97.4
_202_jess	4.5	95.5
_209_db	33.1	66.9
_213_javac	6.1	93.9
_222_mpegaudio	1.4	98.6
_227_mtrt	1.4	98.6
_228_jack	15.1	84.9
Average	9.2	90.8

Table 4-5. Time Breakdown for SPECJVM98 Programs.

Program	System Time (%)	User Time (%)
WebL	54.0	46.0
Cloudscape	33.9	66.1
Mercator	92.9	7.1

Table 4-6. Time Breakdown for Sample Applications.

In summary, the three examples presented demonstrate HBench:Java's ability to predict real applications' performance. The results are especially encouraging since the system vector contains only a small set of system class methods.

To understand why SPECJVM98 was not able to predict application performance correctly, we compared the behaviors of SPECJVM98 programs with those of the three sample applications in terms of time breakdown for user versus system classes. Tables 4-5 and 4-6 show the percentage of time spent in system classes for SPECJVM98 programs and the three sample applications we tested, respectively. These numbers were obtained using the sampling facility of the *hprof* agent included in Sun's JDK1.2.2. As the data

show, the SPEC programs spend most of the time in user classes. Our experience indicates that most real-world Java applications spend a significant amount of time in system classes. Therefore, SPECJVM98 is a poor predictor for the general class of Java applications.

Notice that even though a larger percentage of time goes to user classes for the Cloudscape case, HBench:Java was still able to predict the ratios quite accurately. We suspect that this is because performance of user classes is largely determined by JIT quality. System classes are also compiled by the same JIT, thus performance of a collection of system classes in some way reflects the JIT quality, which applies to user classes as well.

To test this hypothesis, we used HBench:Java to predict the relative performance of the `db` program in the SPECJVM98 suite. Figure 4-7 shows the results. HBench:Java was able to predict the relative running speeds correctly except for the last two JVMs on the X-axis, the SDK3.2 on the Pentium II machine and the JDK1.2.1_O3 on the SPARC workstation. The predicted ratios, however, are quite close to the actual ratios. The results suggest that system classes can be reasonable indicators for JIT quality, but there is room for improvement.

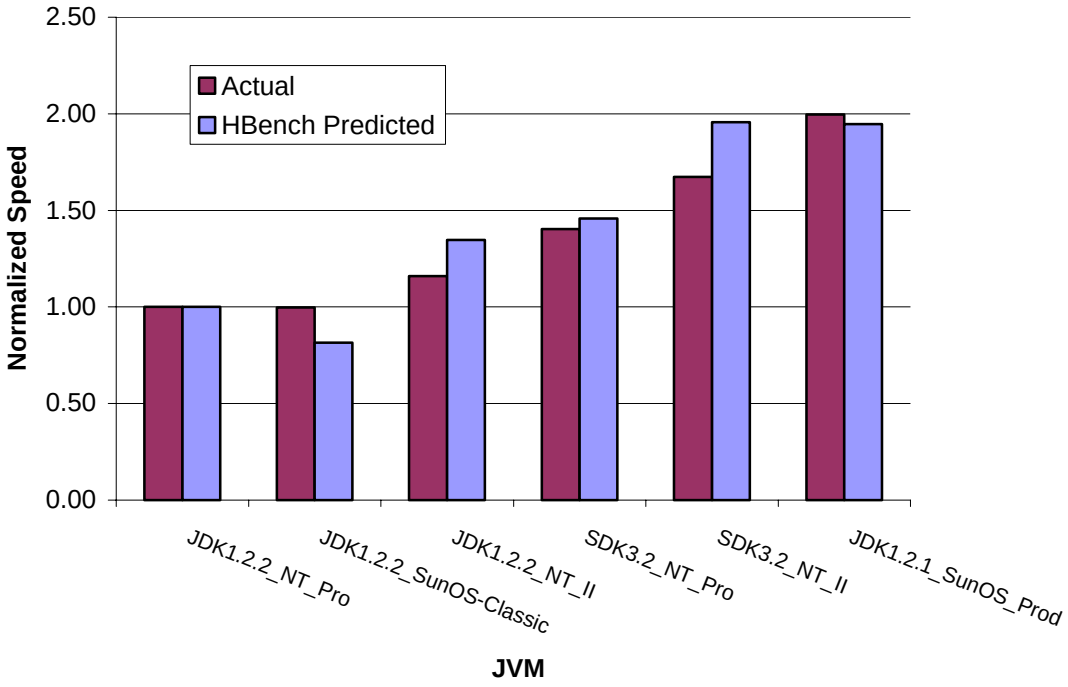


Figure 4-7. Prediction for the db Program of SPECJVM98 Using HBench:Java. The speeds are normalized against the reference JVM. This graph shows that HBench:Java is able to predict the relative running speeds correctly except for the last two JVMs, the SDK3.2 on the Pentium II machine and the JDK1.2.1_O3 on the SPARC workstation.

One possible approach to improving the estimation of JIT quality for user-defined classes is to use results from benchmarks that are user-class bound such as SPECJVM98 and the JavaGrande benchmark and combine them with HBench:Java results to form a prediction. This direction of research is out of the scope of this thesis, but will be explored in future work.

4.4 Discussion

HBench:Java is still in the early stages of its development. Here we identify a few unresolved issues and describe how we plan to address them.

The first issue is the large number of API method calls. We plan to attack this problem by identifying a set of core methods, including methods executed frequently by most applications (such as those in the String class), and methods upon which many other methods are built (such as those in the FileInputStream class). We then plan to analyze method inter-dependencies and derive running time estimates of non-core methods from the running times of the core methods. For instance, a `length()` method typically takes the same time as a `size()` method. We believe that it is acceptable if the estimates of non-core classes are not 100% accurate, since we expect these methods to be infrequently invoked.

A related issue is how do we determine the core set of methods. We currently rely on application traces to identify them because Java is relatively new and we do not yet have enough experience to decide ahead of time what methods are important. As we gain more experience, we will have a better knowledge of what primitives should be included in the core set and what primitives should be estimated. We are confident this would be the situation. HBench:OS (see Chapter 6), for example, does not rely on application traces to determine the primitives, as we have extensive knowledge about how OS primitives are used.

When identifying the set of core methods, there is a tradeoff between simplicity and accuracy. The more methods we include in the core set, the better the prediction becomes, however at the expense of measurement complexity. Since our first goal is to be able to predict the relative order correctly, while keeping the vector as simple as possible, we do not try to account for all contributions. Rather, we look for a subset that gives good but not perfect coverage. It is conceivable that we might miss important

methods in some corner cases, but the HBench:Java methodology should still provide much better predictive power than standard benchmarks.

Another issue is that JIT compilers could alter an application enough that no single application vector could be used across all JVM platforms. Our experience so far indicates that this is not yet a problem. However, we will closely follow this issue as JIT technologies become more advanced.

4.5 Summary

HBench:Java is a vector-based, application-specific benchmarking framework for JVMs. The performance results demonstrate HBench:Java's superiority over traditional benchmarking methods in predicting the performance of real applications and in pinpointing performance problems. By taking the nature of target applications into account and offering fine-grained performance characterizations HBench:Java can provide meaningful metrics to both consumers and developers of JVMs and Java applications.

Chapter 5
HBench:JGC for evaluating GC Performance

In this chapter, we present the techniques for applying the vector-based methodology to the domain of garbage collection and evaluate its effectiveness. We first give an overview on basic concepts of garbage collection in Section 5.1. Section 5.2 describes the design of HBench:JGC in detail. Section 5.3 describes its prototype implementation. Section 5.4 presents experimental results on applying HBench:JGC to predicting GC times. Section 5.5 discusses open issues and future work. Although the HBench:JGC is implemented using Java, the methodology of HBench:JGC described in this paper is applicable to GC implementations for other languages such as Lisp, Scheme, Smalltalk, and C++.

5.1 Introduction

5.1.1 Basic GC Concepts

The garbage collector manages the collection of free space from which new objects are allocated. The free space can be represented as a list of free blocks, a single chunk of contiguous space, or a combination of the two.

When the allocator fails to satisfy an allocation request, it initiates a garbage collection run. A garbage collection run typically starts with a marking phase, when live objects are identified and marked. This phase may be followed by one or more phases (typically called the sweep phases) that free the space occupied by the dead objects, making it available for allocation. A non-copying collector does not move the live objects, whereas a copying collector typically compacts the live objects to one end of the

Attributes of GC Implementations		
Stop-the-world	↔	Concurrent
Sequential	↔	Parallel
Batch	↔	Incremental
Non-generational	↔	Generational

Table 5-1. GC Implementation Techniques

heap in order to create a large contiguous free space at the other end of the heap. Examples of non-copying collectors include the most widely adopted mark-sweep garbage collector [5] and its variants. Examples of copying collectors include the Lisp 2 collector [31], which is a mark-compact collector, and Cheney's two-space copying collector [10]. For a complete treatment of this topic, readers are encouraged to refer to the book by Jones et al. [27].

5.1.2 A GC Implementation Taxonomy

Independent of the GC algorithms (e.g., copying vs. non-copying), we can classify GC implementations according to the four attributes described in Table 5-1. The first attribute represents the axis between stopping all execution for garbage collection and running the collector completely in parallel with program execution [3]. The second attribute describes the internal architecture of the collector itself, whether it is sequential (single-threaded) or parallel (multi-threaded). The third attribute describes the granularity of collection, whether collection occurs in a single, complete pass (batch-oriented) or whether just some of the available memory is reclaimed during each iteration (incremental). The fourth and last attribute distinguishes generational garbage

collectors [34] from non-generational collectors. Generational collectors implement a set of heaps that are cleaned with varying frequency depending on the age of the objects stored in the heap. Each heap corresponds to a different age group.

The four attributes in the taxonomy are largely orthogonal, with a few exceptions. For example, a GC algorithm can be both stop-the-world and parallel, but it cannot be both concurrent and batch mode.

In this thesis we consider only sequential, stop-the-world, batch-mode and non-generational garbage collectors. We chose to start with this type of collector because it involves the fewest variables and thus allows faster prototyping of the analytical models and more controllable experimentation. Furthermore, this type of collector is still in wide use. For example, Sun's standard JDK1.1 and JDK1.2 Java Virtual Machines use this type of collector. Section 5.5 discusses how we envision enhancing our approach to cope with concurrent, parallel, incremental and generational garbage collectors.

5.2 HBench:JGC Design

5.2.1 GC Characterization

Like all memory management systems, a garbage collector implementation supports two primitive operations, namely, object allocation and reclamation.

5.2.1.1 Object Allocation

For a given memory management algorithm, the cost of object allocation is typically determined by the following two factors:

1. the size of the allocation,

2. the state of the heap, such as the number of free blocks and their sizes.

We can represent this cost with a function C_{alloc}(*heap_state*, *allocation_size*). Depending on the memory management algorithm, C_{alloc} carries different forms. In the case of copying garbage collectors, the free space is a contiguous area, and allocation can be implemented by a simple pointer advancement. Therefore, in the case of a copying collector, C_{alloc} is a constant function. In the case of non-copying collectors, such as a non-copying mark and sweep collector, the allocation time depends on the state of the free-block lists maintained by the collector. If we characterize the heap state with simple statistical measures, such as a normal distribution with a given mean and standard deviation, or a uniform distribution with a given range, we can represent C_{alloc} in a concise way. Furthermore, we can measure C_{alloc} using microbenchmarks that initialize the heap according to the statistical measures.

5.2.1.2 Object Reclamation

An interesting aspect of garbage collection performance is that the cost of dead object reclamation depends on the amount of live data on the heap, since the way a garbage collector identifies live objects is to traverse the connected object graph from a set of root objects.

We divide the cost of object reclamation into three parts: the fixed cost (C_{fixed}), the per-live-object cost (C_{live}), and the per-dead-object cost (C_{dead}). C_{fixed} corresponds to the fixed cost associated with a garbage collection run, such as the initialization of data structures. C_{fixed} normally depends only on the heap size. C_{live} is the overhead measured per live object (objects that survive the collection). For non-copying collectors, C_{live} is

typically constant. For copying collectors, C_{live} is a function of the size of live objects, as live objects are compacted (copied) at the end of a collection run. C_{dead} corresponds to the per-object cost of releasing the space of a dead object. In most cases, this involves updating bookkeeping information for the freed object, and thus C_{dead} is usually constant for a given collector algorithm. In summary, the cost of object reclamation can be represented by three functions, C_{fixed}(*heap_size*), C_{live}(*object_size*), and C_{dead}. Let N_l be the distribution function of the sizes of live objects, i.e., $N_l(s)$ is the number of surviving objects with size *s*. Let N_d be the distribution function of dead object sizes. The total cost of garbage collecting a heap of size *h* can then be calculated using the following formula:

$$T_{GC} = C_{fixed}(h) + \sum_s C_{live}(s) \cdot N_l(s) + C_{dead} \sum_s N_d(s) \qquad (1)$$

The above reasoning makes the simplifying assumption that every live object is traversed exactly once during marking. For cases where an object is referenced by several live objects, the object will be visited multiple times by the collector. We characterize this additional cost by adding a second variable, d_i, the fan-in degree of an object, in the per-live-object overhead function C_{live}. The middle term of the formula thus becomes:

$$\sum_s \sum_{d_i} C_{live}(s, d_i) \cdot N_l(s, d_i)$$

The situation is further complicated by the fact that certain copying collectors need to update an object's references, if the objects it points to are copied to a different place. We characterize this additional cost by adding yet another variable, d_o, the fan-out degree of an object, in the per-live-object overhead function C_{live}. The middle term now becomes:

$$\sum_{s}\sum_{d_i}\sum_{d_o} C_{live}(s, d_i, d_o) \cdot N_l(s, d_i, d_o)$$

The difficulty of characterizing object reclamation costs lies in deriving the three cost functions C_{fixed}, C_{live}, and C_{dead} using results from microbenchmarks. Our experience indicates that the simplified formula (1) for estimating GC time works well in practice for a mark-sweep GC algorithm.

5.2.2 Application Characterization

The following metrics describe an application's memory usage behavior:

1. Object allocation rate (both in terms of the number of objects and the number of bytes);

2. Object death rate (both in terms of the number of objects and the number of bytes);

3. Object age (the time an object remains alive);

4. Connectivity of the live object graph, i.e., the number of references to an object (fan-in degree) and the number of references it contains (fan-out degree).

Some of the metrics, such as object allocation rate, can be obtained quite easily. Some other metrics, such as object age, are difficult to measure and can only be estimated using profiling tools.

One significant challenge in characterizing an application's memory behavior is that of GC (and JVM) independence. For example, if we use the number of objects per second as the unit for object allocation speed, it is not portable to other JVM or GC implementations, as this unit is system dependent. To solve this problem, we use objects per bytecode as our basic unit for both object allocation rate and object death rate.

5.2.3 Predicting GC Time

Object allocation cost is an important part of the performance metric of GC systems. It is, however, not directly measurable for a given application. As a first step, this paper focuses on predicting the time the application spends on garbage collection, or the time between the start and finish of a garbage collection run. Unless otherwise specified, GC time refers to the cost of object reclamation, and does not include allocation costs.

The total GC time of an application can be determined by two factors: the number of GC runs and the time for each GC run.

With the knowledge of object allocation rate and object death rate, one can estimate the amount of live data at a given execution point, from which one can then calculate the number of GCs deterministically, assuming a heap that is fixed-size or one whose growth policy is known a priori.

The time for each GC run can be estimated using formula (1) described in Section 5.2.1.2. The total GC time is the sum of times of all individual GC runs.

5.3 HBench:JGC Implementation

The major components of HBench:JGC are: the profiler that traces an application's memory behavior, the set of microbenchmarks whose measurement results form the characterization of the given garbage collection implementation, and finally, the analyzer that estimates the GC time given both application and GC characterizations. The following three subsections describe each component in more detail.

5.3.1 Profiler

We again implement our profiler based on the JVMPI profiling interface provided by Sun Microsystems's JDK 1.2.2. We are interested in the following events: GC start and finish, object allocation, free and move, heap dump and object dump. Object allocation and free events can be used to estimate object lifetimes and the number of free/live objects at a given execution point. Heap dumps help determine the object connectivity such as fan-in and fan-out degrees. Our current implementation includes all the events except heap and object dump.

5.3.2 Microbenchmarks

The goal of microbenchmarking is to measure the fixed and per-object costs of memory reclamation. Our first microbenchmark deals with singular linked list data structures. In the future, we will include microbenchmarks that model more complicated object types with different fan-in and fan-out degrees.

The microbenchmark first populates the heap with an array of linked lists of objects. The size of array, the length of the list, and the object size can all be dynamically

configured with command-line options. The microbenchmark then explicitly invokes garbage collection at three different times:

1. When all objects on the heap are alive;

2. When all objects on the heap are reclaimable, i.e., after the microbenchmark sets the pointers to the heads of the linked lists to null;

3. When the heap is entirely empty, i.e., after the GC following step 2.

To measure C_{fixed}, we run the microbenchmark with different heap sizes, fixing the other two parameters. We then plot the GC times measured in step 3 above against the heap sizes. The resulting regression formula is the approximate function for C_{fixed}.

Similarly, to measure C_{live}, we run the microbenchmark with a varying numbers of objects, fixing the other two parameters. The GC times measured in step 1 above are then plotted against the number of objects for a given object size s and the resulting regression function defines $C_{live}(s)$. Since C_{live} might also depend on object sizes, we again repeat the microbenchmark for different object sizes.

The same process is performed to measure C_{dead}, except that in this case the GC times of step 2 are used.

5.3.3 Analyzer

Given both the application and GC characterizations, the analyzer tries to estimate the time the application spends on garbage collection. The analyzer also needs certain configuration information, such as the heap size, in order to determine the total GC time.

CPU	Memory (MB)	Operating System	JVM Version	GC Algorithm
Pentium Pro 200MHz	128	Windows NT 4.0	1.2.2 Classic	Mostly mark-sweep
Pentium III 550 MHz	256	Windows 2000		
Ultra SPARC IIi 333 MHz	128	Solaris 7		

Table 5-2. Test Platform Configurations.

Note that heap sizes may change dynamically. For example, if the memory system cannot satisfy the allocation request even after a GC, or if the percentage of free space is below a certain threshold, the heap is expanded. The policies as to when and how much to expand the heap should be specified to the analyzer.

5.4 Experimental Results

5.4.1 Experimental Setup

We ran our experiments on Sun Microsystems's JDK1.2.2 classic version on three different machine configurations. Table 5-2 shows the hardware properties.

Sun Microsystems' JDK1.2.2 classic JVM uses a mark-sweep (with compaction) collector. Mark-sweep collection is one of the classical garbage collection algorithms that remains in wide usage today. Due to its conservative nature, it is popular for type-unsafe languages such as C/C++. The collector of the JDK1.2.2 classic JVM is a variation of the classical mark-sweep collector — it occasionally moves live objects around the heap. Although compaction does not occur often for the applications we tested, it does generate some uncertainties that make it harder to predict the GC time.

Application		Allocation (MB)
SPEC	_201_compress	334
	_202_jess	748
	_209_db	224
	_213_javac	518
	_227_mtrt	355
	_228_jack	481
R^2mark		1552

Table 5-3. GC Activity of Test Applications. Data for SPEC is obtained from the benchmark's documentation. The actual numbers appear to differ but the magnitude is the same. Data for R^2mark is obtained from actual measurement on a SPARC IIi 333MHz machine. This table shows that the applications induce extensive GC activities.

We use Java applications included in the SPECJVM98 benchmark suite [53] and the R^2mark benchmark [24] to evaluate the predictive power of our approach. Most SPECJVM98 applications induce extensive GC activities, except _222_mpegaudio, which is excluded from our set of test applications. R^2mark stands for a Radiosity and Ray-tracing based benchmark. It implements a multi-pass rendering algorithm that simulates the lighting of a computer-generated graphic scene. R^2mark is a demanding Java application that also stresses the JVM's memory system. Table 5-3 lists the GC related information about our test applications.

5.4.2 Microbenchmark Results

We report the GC times of the three steps described in Section 5.3.2. Unless otherwise specified, all data points reported in this section are means of 10 runs of the microbenchmark. In most cases, the standard deviation is within 1%.

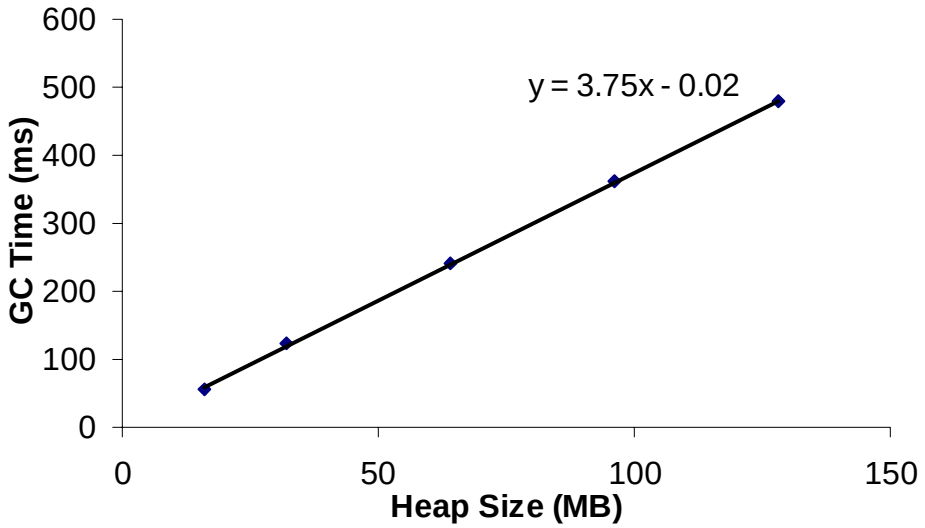


Figure 5-1. GC Time of Empty Heap on Sun SPARC. We use an object size of 28 bytes, and 512 lists each with 512 objects. The number of objects and the size of objects remain fixed as the total heap size varies. This graph shows that the GC times are linearly dependent on the heap size when collecting an empty heap.

5.4.2.1 GC on Empty Heap

Figure 5-1 shows the garbage collection times of an empty heap (see step 3 in section 5.3.2) on the Sun SPARC workstation. The regression formula indicates that GC times of empty heaps are linearly dependent on the size of the heap and that the per-megabyte cost of an empty heap GC for this particular GC implementation is 3.75ms. The y-intercept (0.02) is negligible. We therefore derive the following formula for $C_{fixed}(h)$ (described in Section 5.2.1.2) for this GC algorithm:

$$C_{fixed}(h) = 3.75 \cdot h,$$

where h is the size of the heap in megabytes. The value of the slope (3.75) remains the same (variations within 5%) for different object sizes and numbers of objects.

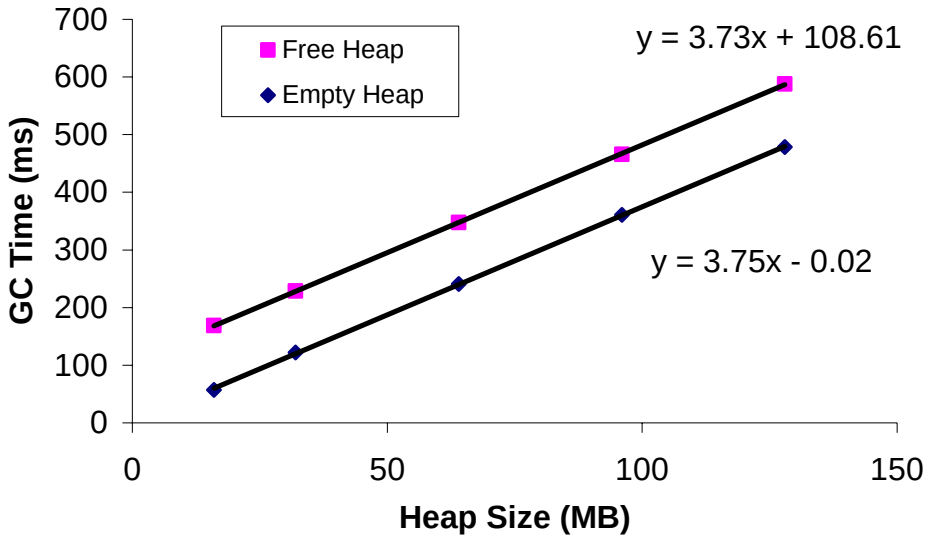


Figure 5-2. GC Time of Fully Reclaimable Heap With Respect to Heap Size on Sun SPARC. We use an object size of 28 bytes, and 512 lists each with 512 objects. The number of objects and the size of objects remain fixed as the total heap size varies. This graph shows that the collection time of a fully reclaimable heap is also linearly dependent on the heap size. The cumulative cost for dead objects, measured by the distance between the two curves, stays constant while the heap size varies.

Similar results were obtained for the other machine configurations, albeit with a different slope value.

5.4.2.2 GC on Fully Reclaimable Heap

Figure 5-2 shows the garbage collection times of a fully reclaimable heap (see step 2 in section 5.3.2). The GC time again shows a linear dependence on the size of the heap, and the slope value (3.73) is close to the slope value of C_{fixed} (3.75). If we remove the fixed cost $C_{fixed}(h)$, the remaining time is essentially independent of heap size. Since all objects on the heap are free and are reclaimed by the collector, this remaining time, when divided by the number of dead objects, represents the per-dead-object cost C_{dead}. In

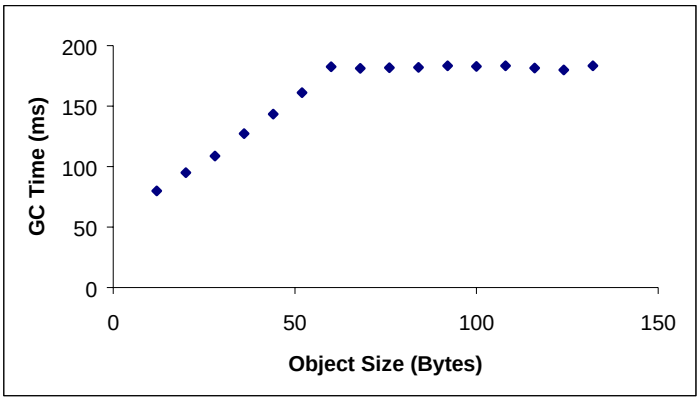
this particular case, C_{dead} takes on a value of 108.6/(512*512), or 0.4 ns/object. Again, similar results are observed from runs on the other machine configurations.

Theoretically, C_{dead} is independent of object size, since dead objects are neither scanned nor copied. However, to our surprise, our measurements suggest that C_{dead} is indeed dependent on object size. Figure 5-3(a) shows the results on the Sun SPARC workstation. The GC time seems to grow as the object size increases, until the object size hits 60 bytes, and stays at around 180ms thereafter. Similar dependence patterns are observed on the Pentium Pro and Pentium III machines, as shown in Figures 5-3(b) and 3(c), respectively. In both cases, C_{dead} is independent of object size, except when the object size is less than 28 bytes.

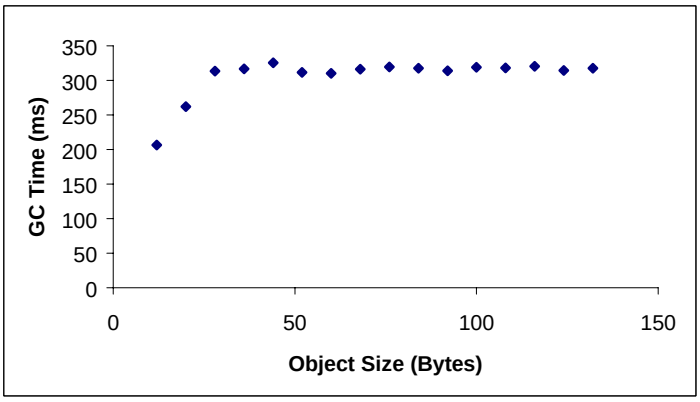
We hypothesize that the dependence on the object size is due to memory cache effects. More specifically, we believe that the size dependence threshold (60 bytes for Ultra SPARC family chips and 28 bytes for Pentium family chips) is determined by the L2 cache-line size of the processor[3]. The L2 cache-line size is 64 bytes for the Ultra SPARC IIi processor, and 32 bytes for the Pentium processor family. When the garbage collector cleans up dead objects, it strides through the heap, inspecting each dead object's header and updates bookkeeping information of free spaces. Notice that normally only the dead object's header is accessed; the content of the object is not touched. For the processors included in our experiments (and for most processor types), the entire heap does not fit in the L2 cache. Consequently, for objects larger than the cache line size, each read on a dead object's header results in a cache miss in both L1 and L2 caches,

requiring the processor to fetch the cache line containing the object header from the main memory. Moreover, this cost is independent of object size. This explains why C_{dead} is constant for objects larger than 60 bytes on the Ultra SPARC IIi workstation, and for objects larger than 28 bytes on the Pentium family machines. For objects smaller than the cache line size, a cache line can pack more than one object. Thus the cost of memory read is amortized across multiple objects accesses. The smaller the object, the more the amortization is. This explains the linear dependence on object size for objects smaller than the cache line size for both the Ultra SPARC IIi workstation and the Pentium family machines.

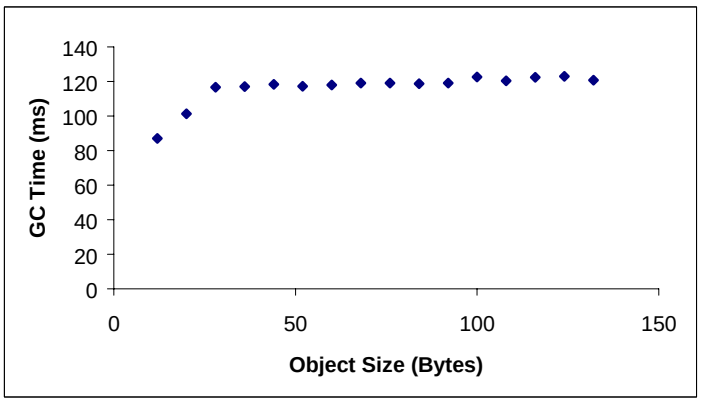
[3] The object size here does not include the size of object header, which is 4 bytes for the garbage collector tested.



(a). Results on 333MHz Sun Ulatra SPARC IIi



(b). Results on Pentium Pro



(c). Results on Pentium III

Figure 5-3. GC Time (Excluding Fixed Overhead) of Fully Reclaimable Heap with Respect to Object Size. The GC time is calculated from the regression formula as shown in Figure 5-2. The cost of dead object grows linearly with the object size up to a certain threshold size, and stays constant afterwards.

To verify our hypothesis, we used the performance counters on the Ultra SPARC IIi chip to record the number of memory-read accesses during garbage collection. We accomplished this with the help of the perfmon tool [43]. The perfmon tool provides a device driver through which user level programs can access the performance counters using standard system call APIs such as `open()` and `ioctl()`. We implemented a profiler that enables the performance counters to monitor the appropriate events at the start of a GC run and reads the counter value at the end of the GC. We then ran the microbenchmarks on the Java Virtual Machine with the profiler attached. We recorded the counts for three types of memory events: L1 data cache read, L1 data cash read hit, L2 cache hit, denoted as $N_{L1\_cache\_read\_total}$, $N_{L1\_cahce\_read\_miss}$, and $N_{L2\_cache\_read\_hits}$, respectively. The memory-read accesses can then be calculated using the following simple formula:

$$\begin{aligned} N_{memory\_read} &= N_{L2\_cache\_read\_total} - N_{L2\_cache\_read\_hits} \\ &= (N_{L1\_cache\_read\_total} - N_{L1\_cache\_read\_miss}) - N_{L2\_cache\_read\_hits} \end{aligned}$$

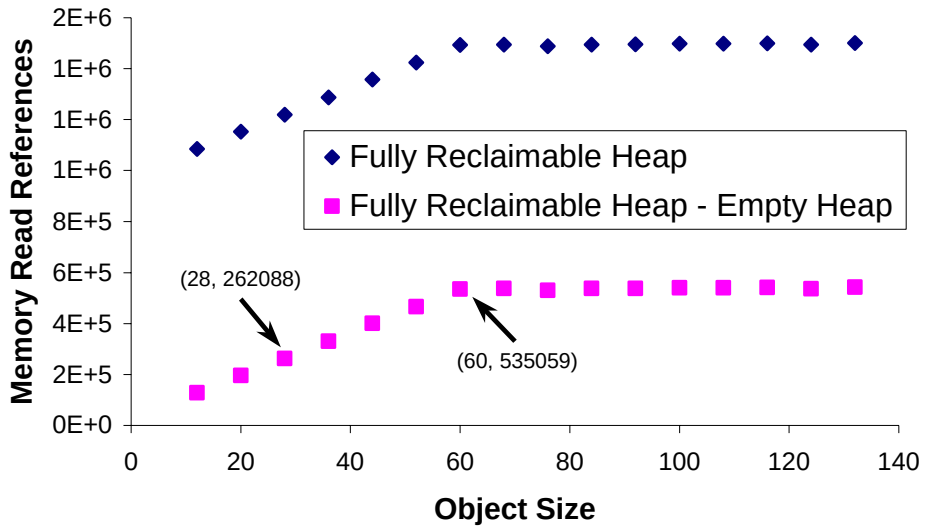


Figure 5-4. Memory-Read Reference Counts as a Function of Object Sizes on 333MHz Sun Ultra SPARC IIi. The top data series represent the case for a reclaimable heap; the bottom data series exclude the counts for the empty heap.

Figure 5-4 shows the memory-read reference counts. The top series represent the memory reference counts for the case of a reclaimable heap (stage 2 in Section 5.3.2). The bottom series represent the memory reference counts minus those for the empty heap case (stage 1 in Section 5.3.2). As one can see, the shape of the data series matches exactly with that in Figure 5-3(a), validating our hypothesis. In addition, the memory reference counts (excluding counts for the empty heap case, shown in the bottom data series in Figure 5-4) for objects size of 28 bytes (32 bytes including the object header) is 262,088, about one half of 535059, the counts for object size of 60 bytes (64 bytes including the object header).

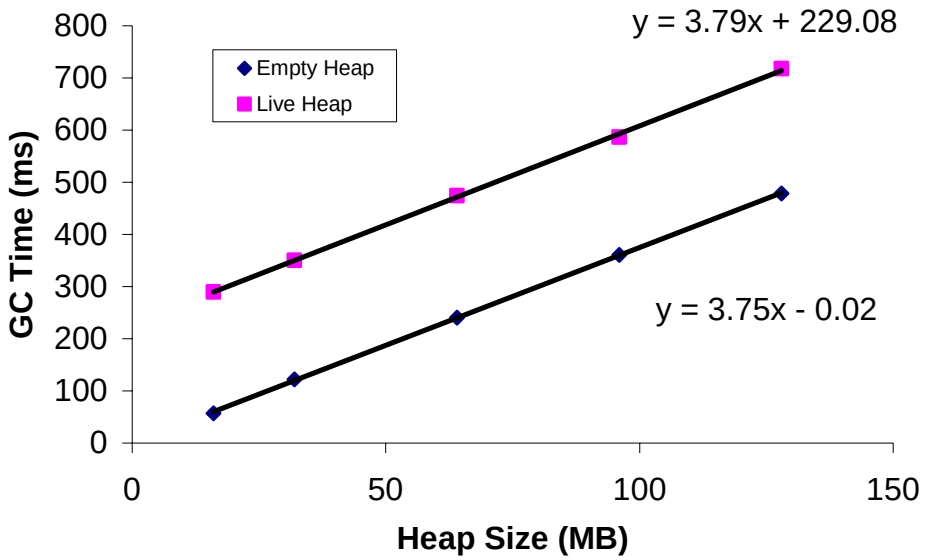
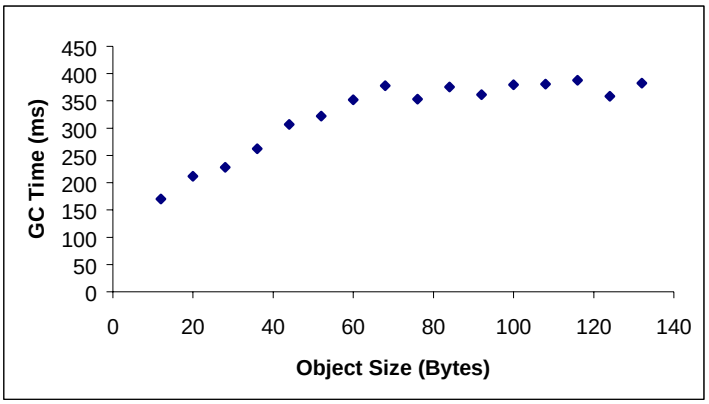


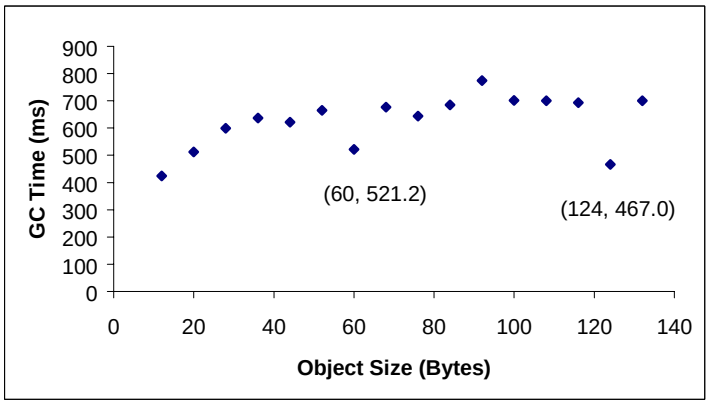
Figure 5-5. GC Time of Fully Live Heap with Respect to Heap Size on Sun SPARC. We use an object size of 28 bytes, and 512 lists each with 512 objects. The number of objects and the size of objects remain fixed as the total heap size varies. This graph shows that the collection time of a fully live heap is also linearly dependent on the heap size. The cumulative cost for live objects, measured by the distance between the two curves, stays constant while the heap size varies.

5.4.2.3 GC on Fully Live Heap

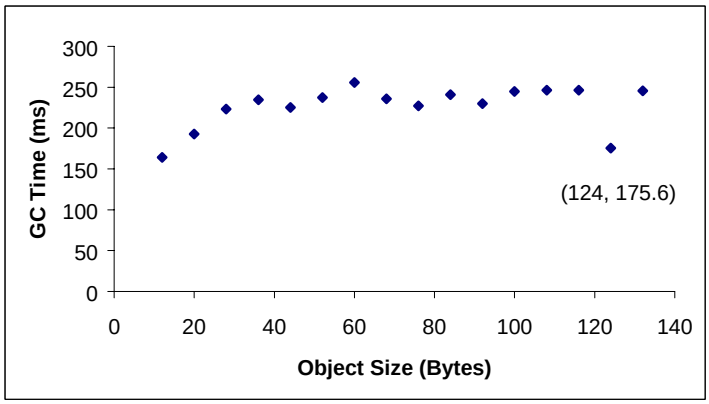
Figure 5-5 shows the garbage collection times of a fully live heap (see step 1 in section 5.3.2). In this case, all objects on the heap are live and survive the garbage collection. Similar to the case of a fully reclaimable heap, the GC time shows a linear dependence on the size of the heap. If we exclude the fixed cost C_{fixed}, the remaining time is independent of heap size. The GC time, when divided by the number of total objects on the heap, yields the per-live-object cost C_{live}. In this particular case, C_{live} takes on a value of 229.1/(512×512), or about 0.9ns/object. Again similar results are observed from runs on other machine configurations.



(a). Results on Sun SPARC



(b). Results on Pentium Pro



(c). Results on Pentium III

Figure 5-6. GC Time (Excluding Fixed Overhead) of Fully Live Heap with Respect to Object Size. The GC time is calculated from the regression formula as shown in Figure 5-5. Similar to the case of a fully reclaimable heap, the cost of live object grows linearly with the object size up to a certain threshold size, and stays constant afterwards.

Figures 5-6(a), 5-6(b) and 5-6(c) show C_{live} as a function of object size on the Sun SPARC workstation, the Pentium Pro machine, and the Pentium III machine, respectively. We observe patterns similar to those of the fully reclaimable heap case, albeit with much larger variations. For the Sun SPARC workstation case, the value of C_{live} seems to grow linearly as the object size increases, until the object size hits 60 bytes and stays at approximately 380ms thereafter. For the Pentium Pro machine case, the value of C_{live} seems to oscillate between 600ms and 700ms after the object size hits 28 bytes. Similarly, for the Pentium III machine case, the value of C_{live} oscillates between 220ms and 250ms after the object size hits 28 bytes.

Similar to the case of a fully reclaimable heap, the memory-read reference counts obtained from the performance counters explain the size dependence of C_{live}, as shown in Figure 5-7. Interestingly, the memory access counts also show an oscillating pattern for objects larger than 60 bytes. Since there is no reason that cleaning up a 84-byte object will take more memory accesses than cleaning up a 92-byte object, we believe that the oscillation observed in Figure 5-6(a) might be due to memory cache effects such as conflict misses in the L2 cache. This effect is also detected with two anomalous data points for the Pentium Pro configuration: at object sizes of 60 bytes and 124 bytes. There is also a similar anomalous data point for the Pentium III at object size of 124 bytes in Figures 5-6(b) and 5-6(c). One explanation for the fact that this variation exists only in the case of live objects (and not dead objects) is that live object manipulation requires looking beyond the object header for pointers to other objects. These extra memory access activities induce more complicated reference patterns of the memory cache, resulting in larger variations in GC times.

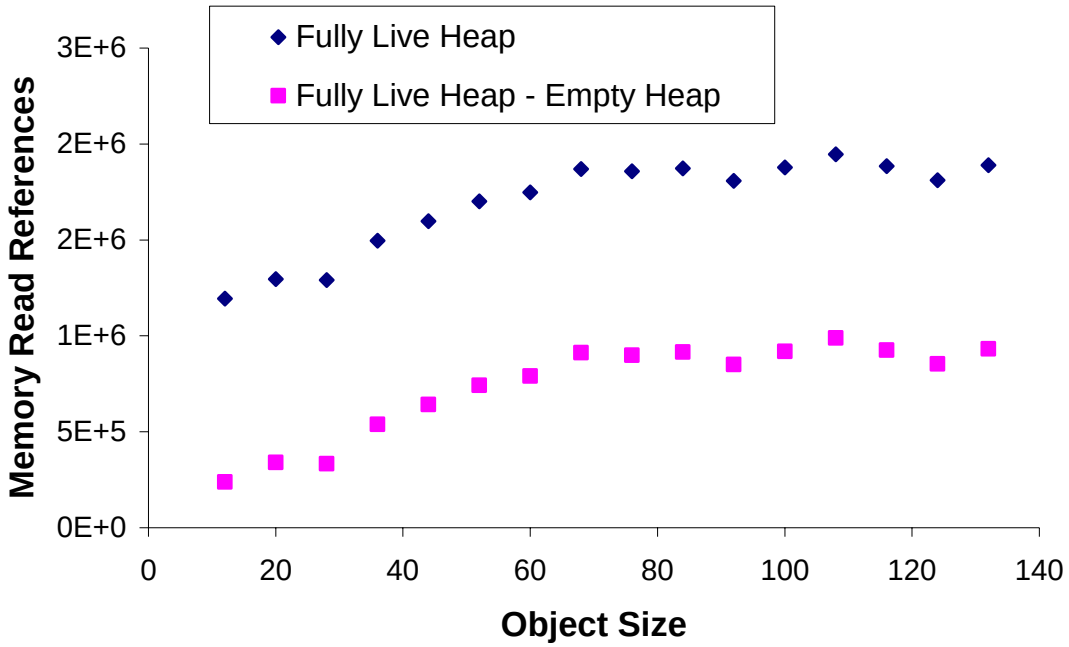


Figure 5-7. Memory-Read Reference Counts as a Function of Object Sizes on 333MHz Sun Ultra SPARC IIi. The top data series represent the case for a fully live heap; the bottom data series exclude the counts for the empty heap.

5.4.3 Predicting GC Time

In this section we demonstrate how the microbenchmark results can be used to predict garbage collection time for a given Java application.

First, we calculate the values of the three functions that characterize a GC algorithm, namely, C_{fixed}, C_{live}, and C_{dead}. Table 5-4 shows the coefficient values of the three functions for the JVM on the Sun SPARC workstation. For objects with size larger than 132 bytes, the values for 132 bytes are used.

Next we obtain characterizations of the applications' memory behavior. Our current profiler implementation generates information such as the number of live objects, the number of dead objects, and the object size distribution. Assuming that live and dead

Object Size	C_{fixed} Per MB	C_{live} Per Object	C_{dead} Per Object
12	3.75	7.04E-04	3.02E-04
20	3.75	7.51E-04	3.49E-04
28	3.75	8.67E-04	4.03E-04
36	3.75	9.55E-04	4.71E-04
44	3.75	1.07E-03	5.49E-04
52	3.75	1.24E-03	6.15E-04
60	3.75	1.30E-03	6.83E-04
68	3.75	1.38E-03	6.85E-04
76	3.75	1.44E-03	6.83E-04
84	3.75	1.41E-03	6.85E-04
92	3.75	1.59E-03	6.85E-04
100	3.75	1.40E-03	6.82E-04
108	3.75	1.33E-03	6.87E-04
116	3.75	1.40E-03	6.91E-04
124	3.75	1.33E-03	6.92E-04
132	3.75	1.46E-03	7.17E-04

Table 5-4. GC Characteristics on 333MHz Ultra SPARC IIi

objects have the same size distribution, we can approximate the GC time function T_{GC} (section 5.2.1.4) with the following formula

$$T_{GC} = C_{fixed}(h) + L \cdot \sum_{s} C_{live}(s) \cdot n(s) + D \cdot \sum_{s} C_{dead}(s) \cdot n(s)$$

where $n(s)$ is the normalized object size distribution function, i.e., $n(12)$ is the percentage of objects with size equal to 12 bytes, L is the number of live objects and D is the number of dead objects. Figure 5-8 shows the accumulative object size distribution function for the test applications. Applications such as `db` and `mtrt` are dominated by one

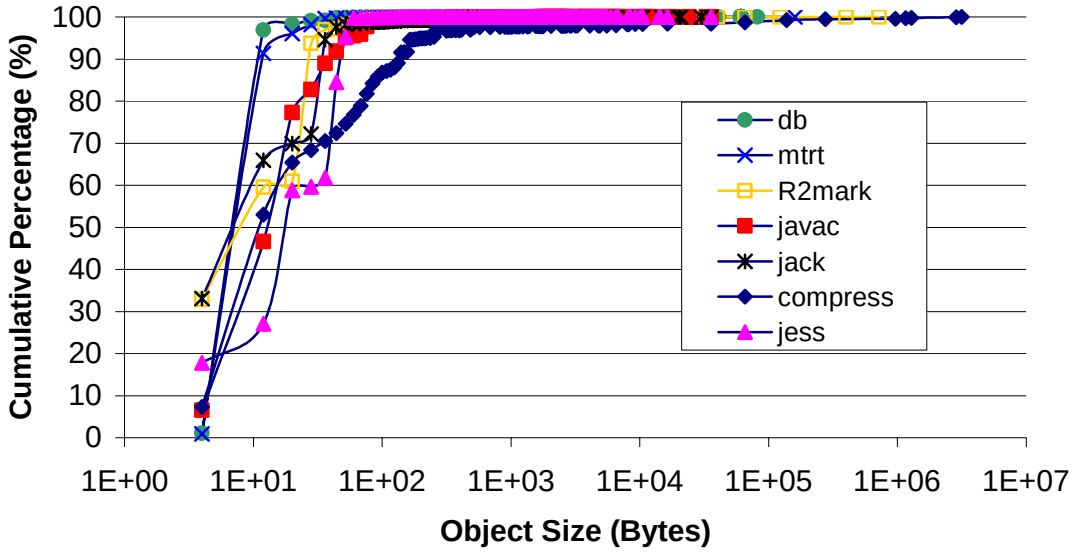


Figure 5-8. Cumulative Object Size Distribution in Number of Objects. This graph shows that the applications differ in the sizes of objects they create. Overall, the majority of objects are small, i.e., less than 100 bytes, except for `compress`.

object size, whereas other applications use multiple object sizes. In general, the majority (more than 90%) of objects are small, i.e., less than 100 bytes, except for compress.

So far our formula has not taken into consideration the cost of the occasional copying performed by the collector. For our test cases, copying only occurred in two applications in four GC invocations (out of a total of over seventy GC invocations). Three of those four GC invocations were explicit garbage collections made by the application, which trigger unnecessary copying. Currently we approximate this copying overhead by dividing the number of bytes copied over the memory bandwidth, and we use the actual number of bytes copied. In the future, we will enhance our analyzer to estimate this information from the application memory characterization, assuming that the algorithm that decides when to perform a copy is known. We will also explore

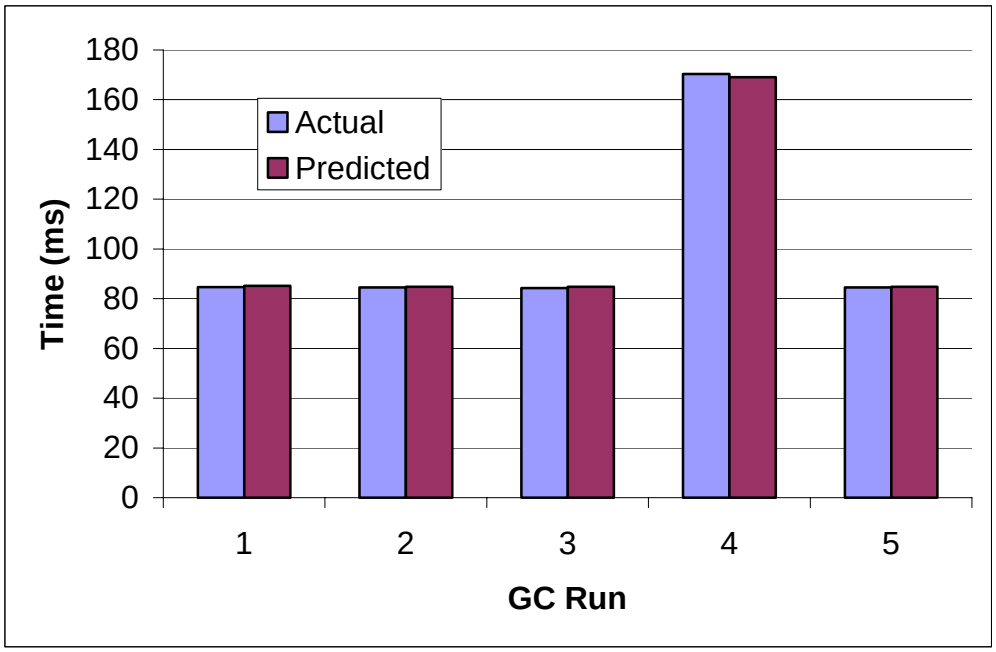
techniques to design microbenchmarks that would trigger a copy and measure the cost directly.

Figure 5-9 shows the predicted versus actual GC running times for the six SPEC applications on the Sun SPARC workstation. A summary of the percentage time difference between the predicted and the actual GC times is presented in Table 5-5.

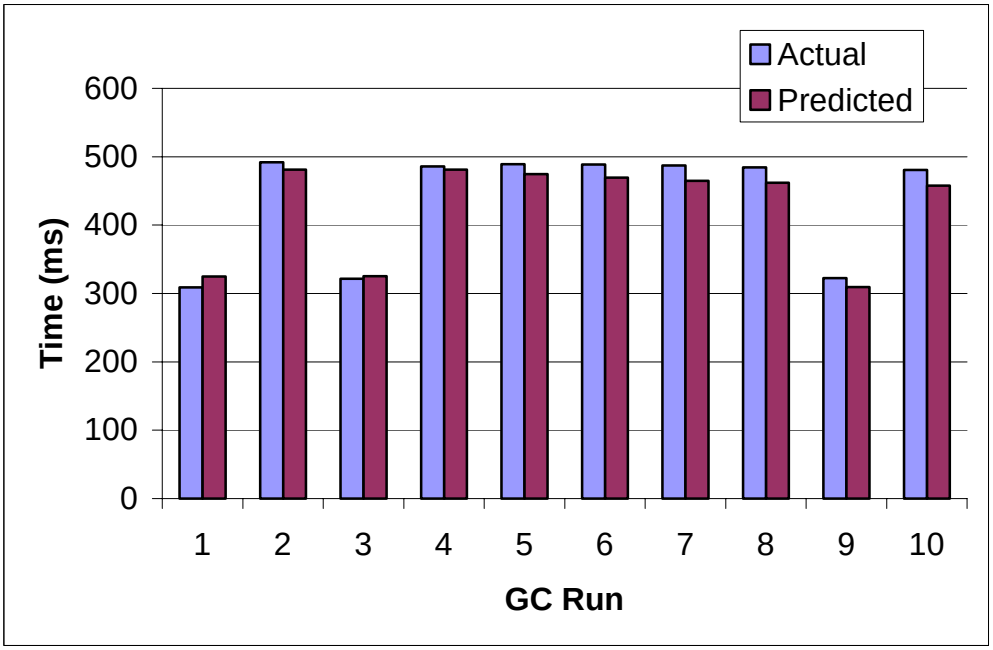
For `compress` (Figure 5-9(a)), there are five garbage collections during the execution of the compress application. The predicted GC times match the actual times quite closely (with 0.2% error rate), showing that our prediction model works well in this case. In the fourth GC run, the collector copied certain live objects to the beginning of the heap, which accounts for the boost in the GC time. The result shows that our approximation on the copying time works well in this case also.

Figures 5-9(b), 5-9(c), 5-9(e) and 5-9(f) show the results for `jess`, `db`, `mtrt` and `jack`, respectively. The predicted times track the actual times quite closely. No copying occurred in these cases.

Figure 5-9(d) shows the results for `javac`. The predicted times track the actual times nicely except for the 3rd, 5th, and 7th GC runs. It turns out that these three GCs were invoked explicitly by the application at times when the heap space had not been exhausted and most objects on the heap were live objects. The explicit GCs also trigger unnecessary copying of live objects. In this case, our approximation on the copying cost does not work well. This might be due to the fact that the approximation does not include the overhead for initiating a copy. Therefore it underestimates the cost in cases when many small objects are copied.

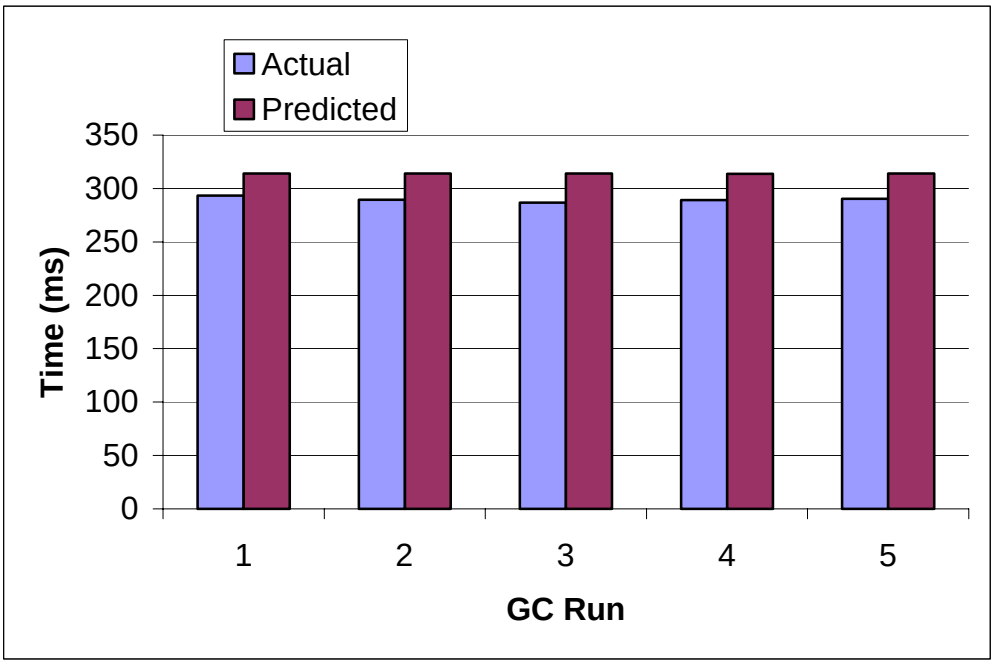


(a). _201_compress. This graph shows that HBench provides a good prediction of actual GC time, with an error rate of 0.2%.

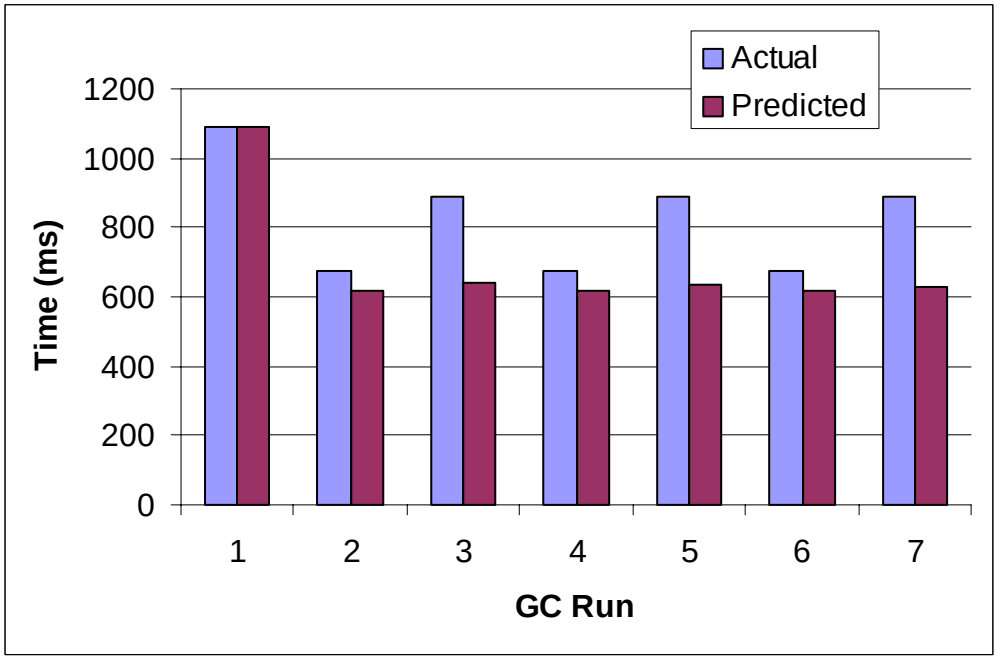


(b). _202_jess. This shows that HBench provides a good prediction of actual GC time, with an error rate of 2.2%.

Figure 5-9. Predicted versus Actual GC Times for SPEC Applications. All tests were run on the Sun SPARC workstation using a heap size of 32MB, except for javac and mtrt, which were run on a heap size of 64MB to eliminate the variation on the number of GCs from different runs.

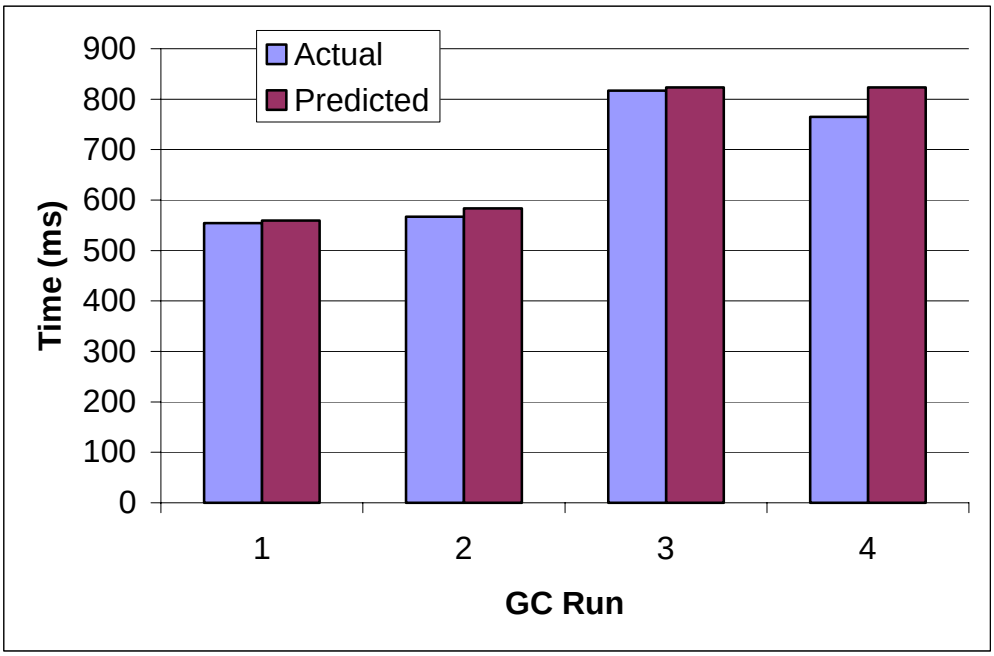


(c). _209_db. This graph shows that HBench provides a good prediction of actual GC time, with an error rate of 8.3%.

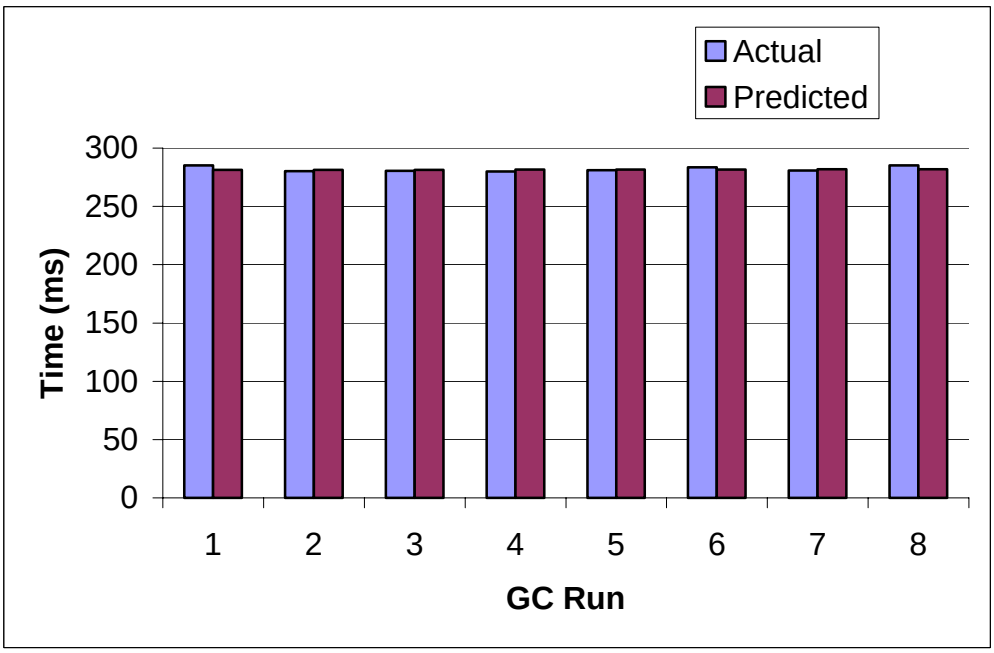


(d). _213_javac. This graph shows that the predicted GC times track closely with the actual times except for runs 3, 5 and 7. These three runs are explicit GCs invoked by the application, in which case the approximation of the copying cost is not sufficiently accurate.

Figure 5-9. Predicted versus Actual GC Times for SPEC Applications. Continued.



(e). _227_mtrt. This graph shows that HBench provides a good prediction of actual GC time, with an error rate of 3.1%.



(f). _228_jack. This graph shows that HBench provides a good prediction of actual GC time, with an error rate of 0.2%.

Figure 5-9. Predicted versus Actual GC Times for SPEC Applications. Continued.

Figure 5-10 shows the results for R²mark. Again, the predicted GC times match actual times fairly closely, except for a few GC runs. One such noticeable exception is the second GC run, where the predicted time is off by almost 50%. A closer look at the data shows that in one particular run of the application, the second GC takes an abnormally long time to finish, pushing the average time higher. In fact, the standard deviation of the actual time of the second GC run is higher than the difference observed between the predicted and actual times. Notice that even though for this particular GC invocation, the variation is large. The average variation is quite small, only 9.9%, as indicated in Table 5-5. Furthermore, the average percentage difference between predicted and actual GC times is only about 2%.

In summary, HBench:JGC is able to predict the actual GC times within 10% for six out of the seven applications (Table 5-5). In the case of javac, the error rate is –6.4% if we disregard the three explicit GCs. The results demonstrate that the vector-based methodology used by HBench:JGC is a promising technique for predicting application performance. In addition, we believe that when equipped with a better profiler and analyzer, the prediction accuracy of HBench:JGC can be improved further.

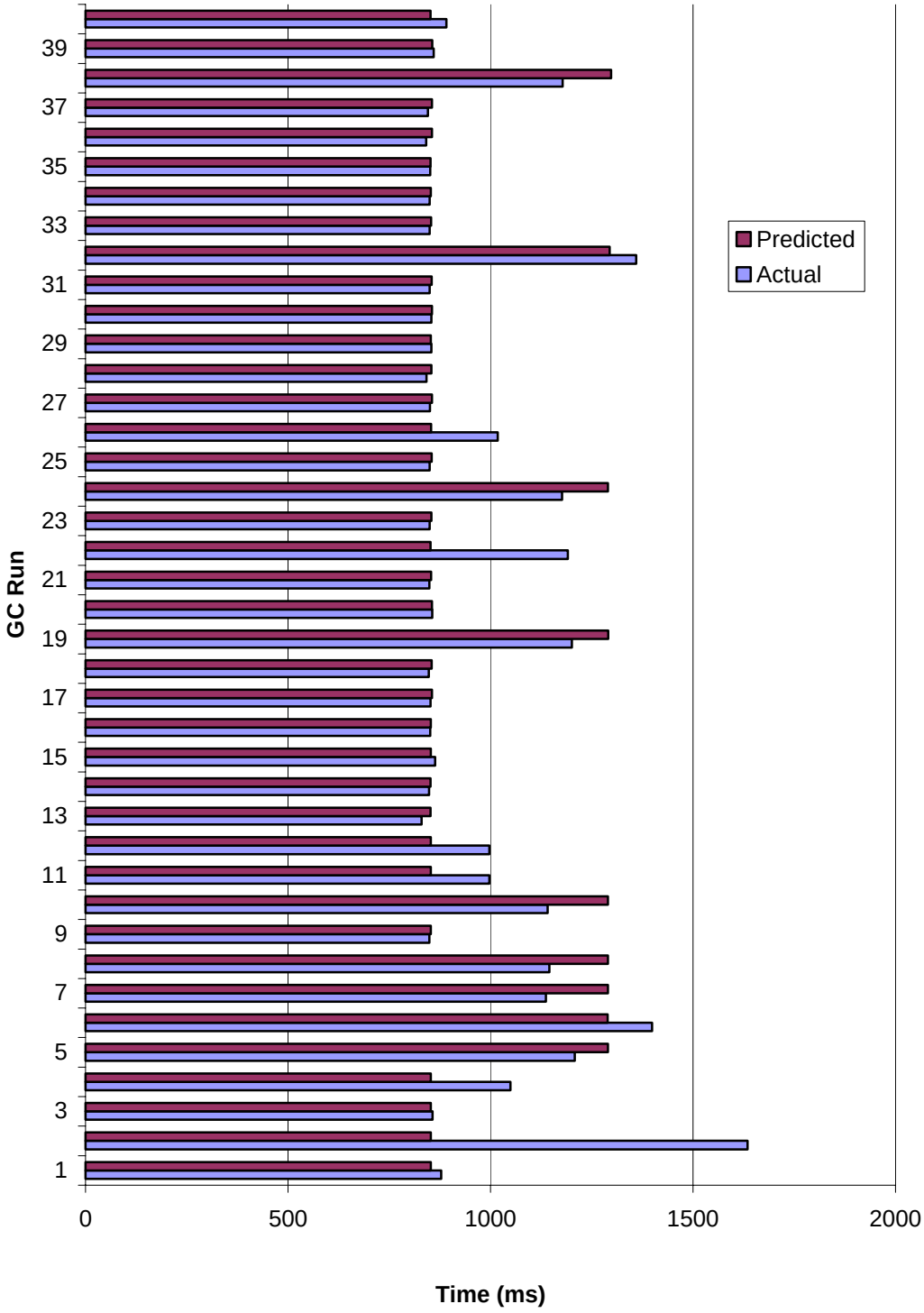


Figure 5-10. Predicted versus Actual GC Times for R^2mark. The test was run on the Sun SPARC workstation using a heap size of 96MB. Data shown are average of 3 runs. This graph shows that HBench provides a good prediction of actual GC time, with an error rate of 1.9%.

Application		Stdev (%)	Time Difference (%)
SPEC	_201_compress	0.5	0.2
	_202_jess	0.4	-2.2
	_209_db	0.8	8.3
	_213_javac	0.5	-15.8(-6.4*)
	_227_mtrt	9.5	3.1
	_228_jack	0.5	-0.2
R^2mark		9.9	-1.9

* Results if we discard 3 explicit GCs.

Table 5-5. Summary of Predicted vs. Actual GC Times

5.5 Discussion and Future Work

In this section we discuss issues that might arise when using HBench:JGC on more sophisticated GC implementations such as those presented in Section 5.1.2, and how we plan to address these issues.

Concurrent garbage collection presents some technical challenges. With concurrent garbage collection, the application can continue to allocate new objects and access objects on the heap while a garbage collection is in process. Measuring the GC time is difficult because the GC time is dispersed in application execution time. We plan to approach this problem in the following way. We run a standard Java application without garbage collection, and then we run the same application with an additional thread that continuously allocates objects and invokes garbage collection. The performance degradation observed when the application is run with the additional GC intensive thread should be a good approximation of the GC time.

Many concurrent collectors are also incremental. Therefore, we will need to estimate the percentage of the heap that is scanned by the collector. In most cases, an incremental collector sets an upper bound on the number of root objects to be processed, from which one can estimate the number of objects on the heap to be scanned.

Predicting the performance of parallel garbage collectors can be potentially difficult because the speed-up of a parallel GC run over its sequential counterpart depends not only on the degree of parallelism, but also on how balanced each thread's load is and the interactions between the threads such as lock contention. Analyzing performance of multi-threaded applications in general is still an active area of research.

To apply HBench:JGC to generational garbage collectors, we model the collector performance for each generation, and then combine them together to form the total GC time. To achieve that, our profiler needs to be enhanced with the capability to estimate the object life expectancy. This can be implemented by sampling the heap at certain time interval *t* and identifying objects that are still alive. Their ages are then incremented by *t*. The age obtained this way can differ from the real age by at most *t*. One might adjust the sampling frequency to attain the degree of accuracy desired. Our analyzer should also be able to predict when objects are promoted to older generations, i.e., it needs to know the age threshold for promotion. Some GC implementations make this knowledge public. For implementations that do not, we need to design our microbenchmark suite such that it can deduce the age threshold by creating and deleting objects at different rates.

Currently, the memory cache effect is included in our cost functions as a function of object size. Our results indicate that in some cases, this simple model might be insufficient. We are investigating ways to model the memory cache hierarchy explicitly.

5.6 Conclusion

HBench:JGC is a vector-based, application-specific benchmarking framework for evaluating garbage collector performance. Our results demonstrate HBench:JGC's unique predictive power. By taking the nature of target applications into account and offering fine-grained performance characterizations of garbage collectors that reflect hardware features such as cache line sizes, HBench:JGC can provide meaningful metrics that help better understand and compare GC performance.

Chapter 6
HBench:OS for Evaluating Operating Systems Performance

In this chapter, we demonstrate how HBench can be applied to the domain of operating systems to analyze and predict performance of kernel-intensive real-world applications.

6.1 Introduction

Operating system performance is critical for many modern server applications such as web servers, as they tend to spend a significant amount of time in kernel.

Brown et al. developed HBench:OS [7], initially to study the performance of different layers in the operating system's structural hierarchy and their inter-dependencies. HBench:OS consists of a collection of microbenchmarks that measure the performance of common system operations and library routines. Originally derived from *lmbench* [39], HBench:OS contains a set of enhancements that make it more suitable for the task of performance prediction. For example, the timing methodology of the original *lmbench* was modified such that the program automatically determines the number of iterations required for accurate measurement with a given timer resolution. A new reporting method, *n*%-trimmed mean, was used for most microbenchmarks to get rid of out-of-band data points. The tests were made more parameterizable to measure bandwidth of small reads/writes that fit in first-level (L1) and second-level (L2) caches. The original context switch measurement included the cost of cache conflict, which often showed large variation. The new revised microbenchmark excluded this cost and only measured the repeatable pure OS overhead for context switches.

6.2 HBench:OS for Predicting Application Performance

Using the improved microbenchmarks of HBench:OS, one can predict application performance with the vector-based methodology. The system vector consists of operating system primitives that characterizes the performance of the system. The application vector entries are counts of invocations to the corresponding system primitives, which represent the load the application places on the operating system. The dot product of the two vectors yields the predicted time spent in kernel. Note that this is kernel time only. HBench:OS is not intended for applications whose performance is dominated by user time.

6.2.1 Application Vector and System Vector Formation

Most operating systems provide utility programs to trace system calls and calls made to the standard libc library by a process and its children. On the Solaris operating system, for example, the tracing tool is called `truss`. We can trace the application running on a certain platform, and then process the trace to obtain the system call counts, which form the application vector. As was done for HBench:Java, the bandwidth metric is converted to a latency metric by calculating per-unit cost. The corresponding application vector entry is then the number of units instead of the number of calls.

The system vector for a given platform can be obtained by running HBench:OS on that platform. Note that the system vector is only acquired once for a given platform and can be combined with different application vectors to predict the application performance on this particular platform.

6.2.2 Summary of Previous Results

Brown used the results of HBench:OS to predict relative performance of the Apache web server on a variety of hardware/OS combinations [6]. He traced the Apache web server serving a single document and derived the application vector from the trace. There were over 100 system calls in the trace, but Brown used a simplified characterization vector containing only six elements: file read, file write, TCP transfer, TCP connection, single handler installation, and "null" system call. Most lightweight system calls were counted as "null" system call, which measured the cost of entering and leaving the kernel and provided a lower bound on the actual system call cost. With this simplified characterization vector, Brown showed that the calculated latency times could correctly rank the machines in the order of relative performance as measured experimentally.

6.2.3 Extension to HBench:OS

One reason that HBench:OS worked well then was that the performance of Apache was dominated by the `read()` system calls to the so-called "scoreboard" file for synchronization. The version of NetBSD that Brown used for his study did not support the `mmap()` interface needed by Apache. Consequently, Apache performed synchronization among its worker processes through reads to the scoreboard file. HBench:OS's ability to predict the performance of the read system call resulted in the correct ranking of Apache's performance on a variety of platforms, even with a simplified characterization vector.

Since that study, both the operating systems and web server workloads have evolved substantially. Current versions of operating systems typically support the `mmap()` interface needed by Apache. Consequently, scoreboard file reads no longer dominate the execution times. A more complex characterization vector is therefore needed for good prediction. On the application side, dynamically generated pages have become common. Therefore, the new application vector must be updated to reflect this change in application behavior. Our goal in this chapter is to extend HBench:OS sufficiently to more accurately predict the performance of today's web servers.

To evaluate the performance of a web server, we need a workload driver that produces web traffic that is sufficiently close to a realistic environment. We decide to use SPEC WEB99 [55]. SPEC WEB99 simulates real-world clients by maintaining a number of simultaneous connections to the server and limiting the bandwidth of each connection to be within the range from 320Kb/s to 400Kb/s. SPEC WEB99 generates both static and dynamic requests. It supports persistent connections specified by HTTP version 1.1, which allows multiple requests to be sent on the same TCP connection. Furthermore, it can also be configured to produce traffic with desired mixes of different types of requests.

The application vector obtained via tracing a web server's processing of a single request, of course, only characterizes the performance of the web server serving that particular request. For this vector to characterize an entire workload generated by SPEC WEB, we need to modify the entries to reflect the average case of the workload. In particular, we modify two entries of the vector: average request size, and average TCP connections per request. The average request size can be calculated based on the

System	Processor	Memory (MB)	OS
Sun-333	Ultra SPARC IIi 333 MHz	256	Solaris 7
Sun-400	Ultra SPARC II 400 MHz	4096	
Intel-550	Pentium III 550 MHz	384	Solaris 8

Table 6-1. Test Machine Configurations.

distribution of request sizes for a given workload. The reason we need to modify the TCP connections per request entry is that for persistent connections, multiple requests can be sent on a single TCP connection. The cost of TCP connection setup is therefore amortized over multiple requests. The average TCP connections per request can be calculated from the percentage of HTTP 1.1 requests and the number of requests per persistent connection.

In some cases, we need to perform manual analysis on the trace to consolidate sequences of system calls into a larger, semantically higher-level system primitive. For instance, a high-level primitive such as forking a child process that in turn executes another program, usually corresponds to many system calls, including `fork()`, `exec()`, and `mmap()` which is called by the loader to load the executable image.

6.3 Experimental Results

6.3.1 Experimental Setup

We run the experiments on the machines listed in Table 6-1. Unless otherwise specified, all timings reported are the average of five runs. The application under consideration is the Apache web server version 1.3.19 [1]. SPEC WEB99 is used to drive the web server.

We decide to use operations per second as the performance metric, as opposed to number of connections used by SPEC WEB99, since we believe that it is less sensitive to the variation of the simulated line speed, which can vary between 320Kb and 400Kb. Furthermore, to make the experiments more controllable, we include only two types of requests: static GET and standard dynamic GET[4]. Therefore, our results should not be compared against published SPEC WEB results.

6.3.2 Static Requests

Table 6-2 lists the application vector for a static HTTP request and the corresponding system vector for the three platforms described in Table 6-1. Figure 6-1 shows the normalized throughputs compared to the predicted throughputs using calculated latencies. As one can see, HBench:OS is able to predict the relative performance ranking correctly. It is interesting to compare the two platforms: Sun-400 and Intel-550. Most primitives of the Intel-550 are about 50% faster than those of the Sun-400, except for TCP bandwidth, TCP connection, and fcntl. The first two primitives happen to account for about 60% of the predicted latency. If one looks only at the performance of these two primitives, one might reach the erroneous conclusion that the Sun-400 is better than the Intel-550. On the other hand, if one accounts for only the other primitives, then one might reach the erroneous conclusion that the Intel-550 is 50% faster than the Sun-400. The fact that HBench uses a weighted average makes sure that the contribution to the total running time made by each primitive is accurately counted. As a

[4] "Standard dynamic GET" requests accounts for more than 40% of the total dynamic requests of the SPEC WEB running configuration, therefore we believe our simplified test configuration reflects important characteristics of real web environments.

Vector Element	Application Vector	Sun-333		Sun-400		Intel-550	
		System Vector (µs)	Total Time (µs)	System Vector (µs)	Total Time (µs)	System Vector (µs)	Total Time (µs)
tcp_transfer	14712	2.49E-2	366.388	1.66E-2	243.690	1.67E-2	245.661
tcp_connect	0.37	425.833	157.558	353.000	130.610	373.833	138.318
tcp_latency	1	119.509	119.509	88.685	88.685	85.082	85.082
mmap_read	14712	6.20E-3	91.259	3.47E-3	51.021	2.23E-3	32.778
open & close	1	31.841	31.841	24.735	24.735	15.196	15.196
fcntl	1	27.349	27.349	22.987	22.987	29.369	29.369
stat	1	24.226	24.226	20.875	20.875	11.447	11.447
mmap	1	19.750	19.750	17.003	17.003	14.760	14.760
signal_handler _install	3	3.196	9.587	2.566	7.699	1.642	4.925
file_write	1	3.312	3.312	2.682	2.682	1.871	1.871
Other	7	3.337	23.360	2.678	18.745	1.869	13.080
Total (µs)		**874.139**		**628.732**		**592.487**	

Table 6-2. System and Application Vectors for Static Request. The request size, 14712, is the average request size calculated from the size distribution function of the SPEC WEB99 workload. Costs of light-weight system calls that do not have corresponding HBench:OS measurement are approximated using the "null" system call, shown above in the "Other" category. TCP related primitives are taken from the measurements on the loopback interface.

result, a correct ordering and a reasonably good estimate of relative performance (ratio) can be achieved.

6.3.3 Dynamically Generated Requests

6.3.3.1 CGI

Dynamically generated pages are commonly seen in today's web servers. The standard approach to implement dynamic pages is through the Common Gateway Interface (CGI). The process that receives a CGI request forks a child process and hands it the request. The child process then calls the `exec()` system call to execute the CGI

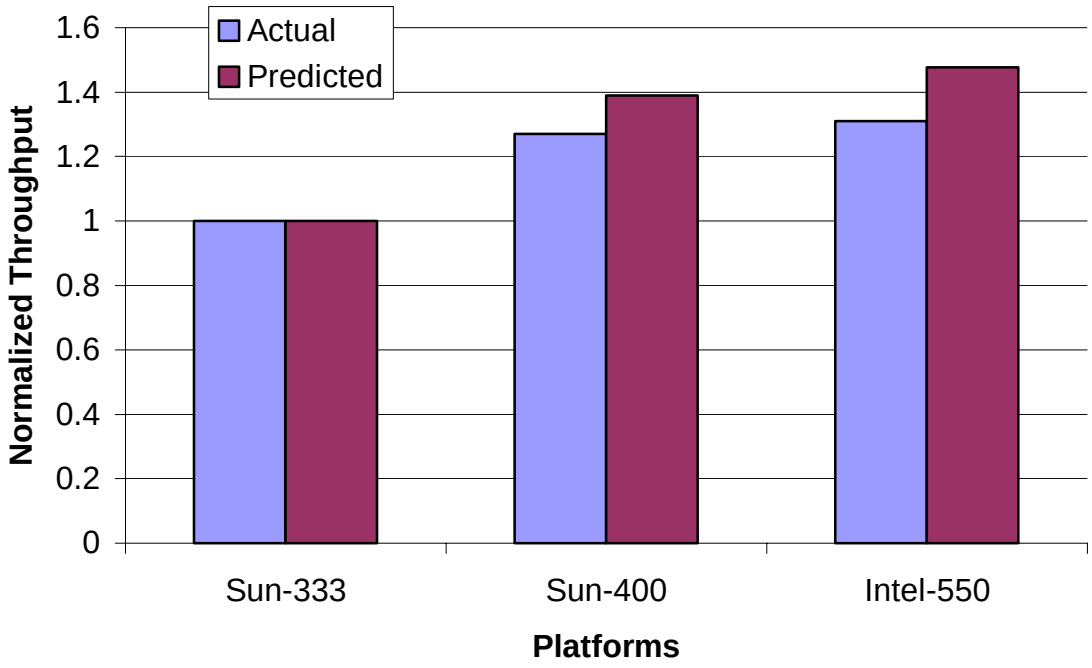


Figure 6-1. Normalized throughputs. Throughputs are normalized against the reference platform Sun-333. This graph shows that HBench is able to predict the relative performance ranking correctly.

program that is specified in the request. The results are returned to the parent process through a pipe.

Table 6-3 and Table 6-4 list the system and application vectors for the parent and child process, respectively. From Table 6-4 we can see that standard CGI requests are an order of magnitude more expensive than static requests, primarily due to the time it takes to fork and execute the CGI application.

6.3.3.2 FastCGI

FastCGI [16] is a newly formed standard that aims to speed up CGI-based dynamic page accesses. The basic idea is to eliminate the costly startup times due to

Vector Element	System Vector (µs)	Application Vector	Total Time (µs)	Percentage (%)
tcp_transfer	1.66E-2	15006	248.559	34.16
pipe_read	1.05E-2	15051	157.388	21.63
tcp_connect	425.833	0.37	130.610	17.95
tcp_latency	88.685	1	88.685	12.19
other	2.678	18	48.202	6.62
fcntl	22.987	1	22.987	3.16
stat	20.875	1	20.875	2.87
signal_handler_install	2.566	3	7.699	1.06
file_write	2.682	1	2.682	0.37
		Total	**727.687**	

Table 6-3. System and Application Vectors for Dynamic CGI Request. Parent process. Measurements were taken on the Sun-400 machine.

Vector Element	System Vector (µs)	Application Vector	Total Time (µs)	Percentage (%)
proc_simple	8256.201	1	8256.201	97.88
file_read	6.46E-3	14712	95.025	1.13
other	2.678	21	56.236	0.67
open & close	24.735	1	24.735	0.29
singal_handler_install	2.566	1	2.566	0.03
		Total	**8434.763**	

Table 6-4. System and Application Vectors for Dynamic CGI Request. Child process. Measurements were taken on the Sun-400 machine. The time to fork and execute the CGI application is approximated using the cost of the primitive that forks and executes a simple dynamically linked program. This table shows that dynamically generated requests are an order of magnitude more expensive than static requests.

`exec()` and `fork()` for every incoming dynamic request. Instead, a persistent FastCGI server process is created when the web server starts up or when the first FastCGI request is received. The process is persistent because it does not terminate after serving the

Vector Element	System Vector (µs)	Application Vector	Total Time (µs)	Percentage (%)
tcp_transfer	1.66E-2	15082	249.818	23.51
socket_rw	1.50E-2	16263	245.132	23.06
tcp_connect	425.833	0.37	130.610	12.29
file_read	6.46E-3	14712	95.025	8.94
tcp_latency	88.685	1	88.685	8.34
stream_connect	79.190	1	79.190	7.45
open & close	24.735	2	49.470	4.65
fcntl	22.987	2	45.975	4.33
stat	20.875	2	41.749	3.93
other	2.678	10	26.779	2.52
signal_handler_install	2.566	3	7.699	0.72
file_write	2.682	1	2.682	0.25
		Total	**1062.814**	

Table 6-5. System and Application Vectors for FastCGI Request. Measurements were taken on the Sun-400 machine.

request. Subsequent FastCGI requests are routed to the server process. There is no costly overhead for forking a process to serve the request. There is, however, additional cost introduced due to the communication between the process that first receives the request and the server process.

Table 6-5 lists the vectors for serving a FastCGI request. As one can see, the elimination of the `fork()` and `exec()` overhead dramatically improved the latency. As expected, the latency is higher than that of the static case because of the extra cost of communicating between the process servicing the request and the FastCGI server process.

6.3.3.3 Predicting Performance Improvements

Using the characterization vectors for the three cases: static, standard CGI, and FastCGI, we can try to predict the performance gains obtained by switching from standard CGI to FastCGI for workloads with different mixes of dynamic and static requests.

Let s_1 be the calculated latency for the original unoptimized case (CGI), and s_2 be the calculated latency for the optimized case (FCGI), then in theory,

$$r = \frac{s_1}{s_2}$$

should be the speedup due to the optimization. However, note that both s_1 and s_2 include only kernel times. The real speedup should be

$$r' = \frac{s_1 + u}{s_2 + u} = \frac{1 + \frac{u}{s_1}}{\frac{s_2}{s_1} + \frac{u}{s_1}} = \frac{1 + \frac{u}{s_1}}{\frac{1}{r} + \frac{u}{s_1}},$$

where u is the user time, assuming that the user-level functionality remains unchanged despite the optimization at the kernel level. When $u/s_1 << 1/r$, the above formula approaches r. This is the case for small r and small u values. However, when r is large, the term u/s_1 can significantly affect the value of r'. In those cases, we need to include u/s_1 in the calculation of r'. u/s_1 is the ratio of user time vs. kernel time in the unoptimized case. In this particular case, the unoptimized case spends about 10%-15% time in user-level. So we use a value of $u/s_1 = 0.1/0.9 = 0.11$ or $u/s_1 = 0.15/0.85 = 0.18$.

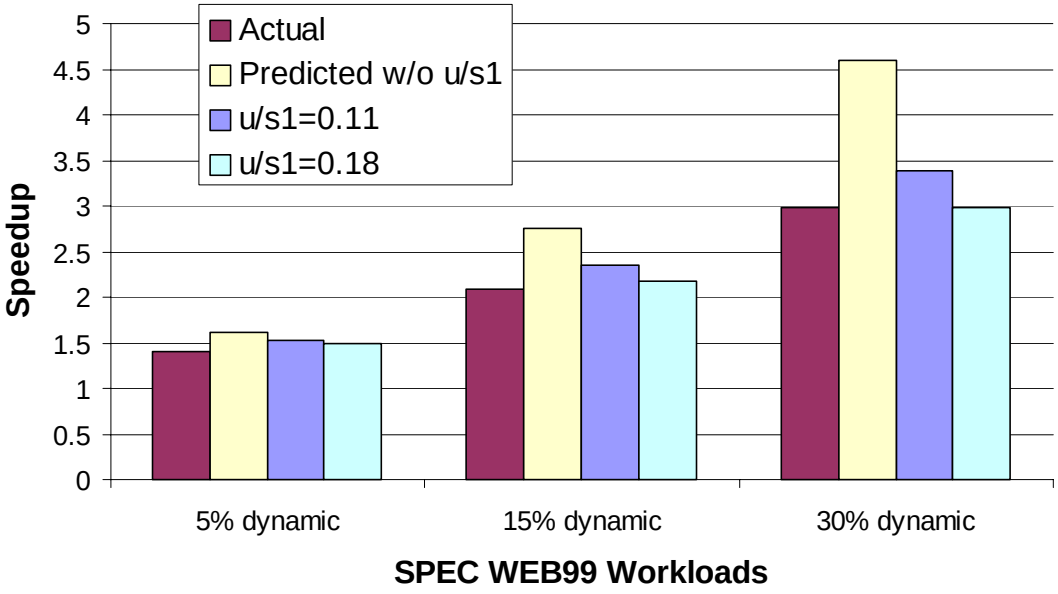


Figure 6-2. Predicted vs. Actual Speedup. Measurements were taken on the Sun-400 machine. This graph shows that HBench is able to predict the performance speedups within 15% margin. Variations of u/s_1 do not significantly affect the accuracy of the prediction.

Figure 6-2 shows the predicted vs. actual speedups for three workload configurations, 5% dynamic (95% static), 15% dynamic (85% static) and 30% dynamic (70% static). Note that the more dynamic requests are included, the larger the speedup. When the user time is not included in the calculation, the predicted speedup deviates significantly from the actual speedup, especially when *r* is large (30% dynamic). With a rough estimate of user vs. system time, the accuracy is dramatically improved. In addition, the graph shows that the estimate of u/s_1 need not be very accurate. Changing the value of u/s_1 from 0.11 to 0.18 (~70% difference) results in less than 15% change in the calculated *r'* in the worst case. Note that the formula presented here applies only to the case where the user time remains unchanged. This does not apply to the case of comparing two different platforms, for example, since the user times would have been different.

6.4 Summary

In summary, we have shown how HBench:OS can be used to analyze performance bottlenecks of complex applications and predict their performance when the application vectors change as a result of optimization.

Chapter 7
Conclusions and Future Work

This thesis proposes a non-traditional approach to performance evaluation that decomposes the benchmarking process into two independent sub-processes: application performance characterization via profiling and system performance characterization via microbenchmarking. Results from the two sub-processes are then linearly combined to form a prediction of the application running time. The decomposition allows applications to participate in the benchmark process, thus the benchmark scores reflect the expected performance of the system in light of the application of interest. As a result, more meaningful comparisons can be made, and better analysis of system and application bottlenecks is possible to provide useful feedback information for future optimization.

This thesis examines three case studies in three different fields of computer sciences that demonstrate the viability of the HBench approach. Section 7.1 summarizes the research findings. Section 7.2 describes the lessons learned, and Section 7.3 explores future research directions.

7.1 Results Summary

In the domain of Java Virtual Machines, we implement a microbenchmark suite that measures the performance of a subset of standard Java system APIs. These results are then used with application profiles to predict realistic Java applications' performance. Results on a variety of JVMs demonstrate HBench:Java's superiority over traditional benchmarks in reflecting real applications' performance and in its ability to pinpoint performance problems.

In the domain of garbage collection, we measure per-object manipulation cost and devise theoretical models to predict GC times. Our microbenchmark results capture architectural features such as memory cache-line size, which are typically excluded from conventional models, and we are able to predict GC times with less than 10% error rate for all but one application.

In the domain of operating systems, we apply the methodology to evaluating the performance of the Apache web server using system calls as primitives. We demonstrate that HBench:OS can be used to analyze performance bottlenecks and to predict performance gains resulting from a common optimization.

7.2 Lessons Learned

The most important decision in designing HBench benchmarks is choosing the interface, i.e., where to draw the line between system and application. In some cases, this is straightforward – for example, in the case of operating systems, it is clear that one should draw the line between the user and kernel address spaces. In some cases, there are multiple choices. For example, in the case of Java Virtual Machines, one could draw the line at the bytecode level or at the system API level. In yet other cases, the choice is not clear. For example, in the case of garbage collection, there is no clearly defined API for the garbage collector, except for object allocation. From our experience, we learned a few guidelines that might help in making a decision in situations where the best choice of interface is not obvious:

1. The interface should be based on well-maintained standard such that the microbenchmarks are portable across different platforms (systems). In cases

where such a standard is not available, e.g., garbage collectors, one might define primitives as parameters of the standard algorithm upon which different implementations are based.

2. The number of primitives defined by the interface must be manageable, i.e., in the order of a few hundreds. Otherwise, it would be too time-consuming to measure each primitive.

3. The performance characterization of the primitives themselves must be deterministic. In other words, the primitives' performance can be measured via repeated invocations and the primitives exhibit the same performance even when invoked in a different context. This is often not true in the presence of heavy hardware or compiler optimization, as in the case of Java bytecode and Just-In-Time compiler.

7.3 Future Research Directions

In the short term, we would like to extend the HBench approach to more domains such as database systems. Database systems resemble operating systems in that they also assume the role of resource management. The difference lies in the interface. Most relational database systems employ a standard query language called SQL [48]. Using SQL one can specify operations such as inserting records, searching a table for records that satisfy certain conditions, and deleting records. A straightforward way of applying HBench to database systems would be to use simple queries as primitive operations and decompose complex queries into these simple primitives. The costs of primitives can be measured and then used to predict the performance of complex queries.

So far, we have discussed benchmark suites for several different domains in the HBench framework. A complex application might span several domains. For example, a Java application might connect to a database backend and execute queries. In such cases, we need to compose the predictions from all domains involved to obtain the total running time prediction. The composition process can be as straightforward as just a summation of the predictions from all domains, or it can get quite complicated. For example, a Java application might incur paging activity, which cannot be directly inferred from the application's profiles. In the future, we would like to explore techniques that would detect such cases automatically and derive application vectors for all domains from the application profile.

Another area for future investigation is program optimization. Application-specific benchmarks offer performance information with finer granularity and richer semantics that could help application and system programmers tune their code for better performance. However, this is largely a manual process. On the other hand, both the compiler and the operating system community have explored techniques of automatic profiled-based optimization [25] [44] [64]. The runtime dynamically optimizes the application code depending on real-time program behavior. We could enhance these techniques with finer-granularity performance characterizations and analysis to improve optimization quality and to direct optimization efforts more effectively.

7.4 Summary

This thesis presented application-specific benchmarking, a non-traditional approach to performance evaluation, based on the principle that systems performance

should be measured in the context of the application of interest to a particular end user. This approach is applied to the domains of Java Virtual Machines, garbage collection, and operating systems. These three case studies demonstrate that application-specific benchmarking is a promising approach that can better predict real application's performance than traditional approaches.

Bibliography

[1] The Apache Software Foundation. http://www.apache.org/.

[2] Arnold, K., and Gosling, J., *The Java Programming Language*. Addison-Wesley, Reading, MA, 1996.

[3] Baker, H. G. Jr., "List Processing in Real Time on a Serial Computer." *Communications of the ACM*, 21(4), pages 280-294, April 1978.

[4] Bershad, B. N., Savage, S., Pardyak, P., and Sirer, E. G., "Extensibility, Safety and Performance in the SPIN Operating System." In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 267-284, Copper Mountain, CO, December 3-6, 1995.

[5] Boehm, H., and Weiser, M., "Garbage Collection in an Uncooperative Environment." *Software–Practice and Experience*, pages 807-820, September 1988.

[6] Brown, A. B., "A Decompositional Approach to Computer System Performance Evaluation." Technical Report TR-03-97, Center for Research in Computing Technology, Harvard University, 1997.

[7] Brown, A. B., and Seltzer, M., "Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture." In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214-224, Seattle, WA, June 15-18, 1997.

[8] Bull, J. M., Smith, L. A., Westhead, M. D., Henty, D. S., and Davey, R. A., "A Methodology for Benchmarking Java Grande Applications." In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 81-88, Palo Alto, CA, June 12-14, 1999.

[9] CaffeineMark. http://www.webfayre.com/pendragon/cm3/runtest.html.

[10] Cheney, C. J., "A Non-Recursive List Compacting Algorithm." *Communications of the ACM*, 13(11), pages 677-678, November 1970.

[11] Cloudscape. http://www.cloudscape.com.

[12] Cohen, J., and Nicolau, A., "Comparison of Compacting Algorithms for Garbage Collection." *ACM Transactions on Programming Languages and Systems*, 5(4), pages 532-553, 1983.

[13] Detlefs, D., Dosser, A., and Zorn, B., "Memory Allocation Costs in Large C and C++ Programs." *Software–Practice and Experience*, 24(6), pages 527-542, June 1994.

[14] Dixit, K. M., "Performance SPECulations – Benchmarks, Friend or Foe." Lecture on SPEC and Benchmarks. In *Seventh International Symposium on High Performance Computer Architecture (HPCA-7)*, Monterrey, Mexico, January 20-24, 2001. Slides available at http://www.csl.cornell.edu/hpca7/presentation.pdf.

[15] Dujmovic, J. J., and Howard, L., "A Method for Generating Benchmark Programs." Technical Report SFSU-CS-TR-00.09, Department of Computer Science, San Francisco State University, June 16, 2000.

[16] FastCGI. http://www.fastcgi.com.

[17] The gcc home page. http://gcc.gnu.org.

[18] Grace, R., *The Benchmark Book*. Prentice Hall, Upper Saddle River, New Jersey, 1996.

[19] Gray, J., *The Benchmark handbook: for database and transaction processing systems*. Second Edition, San Mateo, California; Morgan Kaufmann Publishers, 1993.

[20] Gustafson, J. L., and Snell, Q. O., "HINT: A New Way To Measure Computer Performance." In *Proceedings of the HICSS-28 Conference*, Wailela, Maui, Hawaii, January 3-6, 1995.

[21] The gzip home page. http://www.gzip.org.

[22] Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., and Wotler, J., "The Performance of µ-Kernel-Based Systems." In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 66-77, St. Malo, France, October 5-8, 1997.

[23] Hennessy, J., and Patterson, D., *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, California, 1990.

[24] Hitoshi, Y., Maeda, A., and Kobayashi, H., "Developing a practical parallel multi-pass renderer in Java and C++ - Toward a Grande Application in Java." In *Proceedings of the ACM 2000 Conference on Java Grande*, San Francisco, CA, June 3-4, 2000.

[25] HotSpot. http://java.sun.com/products/hotspot/.

[26] Jones, M., and Regehr, J., "The Problems You're Having May Not Be the Problems You Think You're Having: Results from a Latency Study of Windows NT." In *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems (HotOS VII)*, pages 96-102, Rio Rico, AZ, March 29-30, 1999.

[27] Jones, R., and Lins, R. D., *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Son Ltd, New York, 1996.

[28] JVMDI, Java Virtual Machine Debugger Interface. http://java.sun.com/products/jdk/1.2/docs/guide/jvmdi/index.html.

[29] JVMPI, Java Virtual Machine Profiling Interface. http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/index.html.

[30] Katcher, J., "PostMark: A New File System Benchmark." Technical Report TR3022, Network Appliance. http://www.netapp.com/tech_library/3022.html.

[31] Knuth, D. E., *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Second Edition, Addison Wesley, Reading, MA, 1973.

[32] Lazowska, E. D., Zohorjan, J., Graham, G. S., and Sevcik, K. C., *Quantitative System Performance, Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[33] Liang, S., and Viswanathan, D., "Comprehensive Profiling Support in the Java Virtual Machine." In *5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)*, pages 229-240, San Diego, CA, May 3-7, 1999.

[34] Lieberman, H., and Hewitt, C., "A Real-Time Garbage Collector Based on the Lifetimes of Objects." *Communications of the ACM*, 26(6), pages 419-429, June 1983.

[35] Lindholm, T., and Yellin, F., *The Java Virtual Machine Specification*. Second Edition, Reading, MA, Addison-Wesley, 1999.

[36] Manley, S., and Seltzer, M., "Web Facts and Fantasy." In *Proceedings of the Symposium on Internet Technologies and Systems*, pages 125-134, Monterey, CA, December 8-11, 1997.

[37] Manley, S., Courage, M., and Seltzer, M., "A Self-Scaling and Self-Configuring Benchmark for Web Servers." Technical Report TR-17-97, Center for Research in Computing Technology, Harvard University, 1997.

[38] Mathew, J. A., Coddington, P. D., and Hawick, K. A., "Analysis and Development of Java Grande Benchmarks." In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 72-80, Palo Alto, CA, June 12-14, 1999.

[39] McVoy, L., and Staelin, C., "lmbench: Portable Tools for Performance Analysis." In Proceedings of the 1996 USENIX Technical Conference, pages 279-294, San Diego, CA, January 22-26, 1996.

[40] Mercator. http://www.research.digital.com/SRC/mercator/.

[41] Mogul, J. C. "Brittle Metrics in Operating Systems Research." In *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems (HotOS VII)*, pages 90-95, Rio Rico, AZ, March 29-30, 1999.

[42] Ousterhout, J., "Why Aren't Operating Systems Getting Faster As Fast As Hardware?" In Proceedings of the 1990 Summer USENIX Technical Conference, pages 247-256, Anaheim, CA, June 11-15, 1990.

[43] Perfmon Performance Monitoring Tool. http://www.cse.msu.edu/~enbody/perfmon.html.

[44] Pu, C., Autrey, T., Black, A., Consel, C., Cowan, C., Inouye, J., Kethana, L., Walpole, J., and Zhang, K., "Optimistic Incremental Specialization: Streamlining a Commercial Operating System." In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 15-26, Copper Mountain Resort, CO, December 3-6, 1995.

[45] Saavedra-Barrera, R. H., Smith, A. J., and Miya, E., "Machine Characterization Based on an Abstract High-Level Language Machine." *IEEE Transactions on Computer*, 38(12), December 1989, 1659-1679.

[46] Saavedra-Barrera, R. H., and Smith, A. J., "Analysis of Benchmark Characteristics and Benchmark Performance Prediction." *ACM Transactions on Computer Systems*, 14(4), November 1996, 344-384.

[47] Saito, Y., Mogul, J., and Verghese, B., "A Usenet Performance Study." http://www.research.digital.com/wrl/projects/newsbench/usenet.ps. November, 1998.

[48] Silberschatz, A., Korth, H., Sudarshan, S., *Database System Concepts*. Third Edition, McGraw-Hill, 1998.

[49] Smith, F., and Morrisett, G., "Comparing Mostly-Copying and Mark-Sweep Conservative Collection." In *Proceedings of the 1998 International Symposium on Memory Management*, pages 68-78, Vancouver Canada, October 17-19, 1998.

[50] Smith, K., *Workload-Specific File System Benchmarks*. Ph.D. thesis, Harvard University, 2000.

[51] Snelling, D. A., "Philosophical Perspective on Performance Measurement." *Computer Benchmarks*, Dongarra, J., and Gentzsch, W., editors, North Holland, pages 97-103, Amsterdam, 1993.

[52] SPEC CPU2000. http://www.spec.org/osg/cpu2000/. As of July 2000, SPEC CPU95 is officially retired by SPEC CPU2000. Information about SPEC CPU95 can be found at http://www.spec.org/osg/cpu95/.

[53] SPEC JVM98 Benchmarks. August 1998 release, http://www.spec.org/osg/jvm98/.

[54] SPEC MAIL2001. http://www.spec.org/osg/mail2001/.

[55] SPEC WEB99 Benchmark. http://www.spec.org/osg/web99/. As of April 2000, SPEC WEB96 is officially retired by SPEC WEB99. Information about SPEC WEB96 can be found at http://www.spec.org/osg/web96/.

[56] Standard Performance Evaluation Corporation. http://www.spec.org/.

[57] Transaction Processing Performance Council. http://www.tpc.org.

[58] Venners, B., *Inside the Java 2 Virtual Machine*. Second Edition, McGraw-Hill, 1999.

[59] VolanoMark. http://www.volano.com/benchmarks.html.

[60] Wall, L., Christiansen T., and Schwartz R. L., *Programming Perl*. Second Edition, O'Reilly & Associates, Inc., Sebastopol, CA, 1996.

[61] WebL. http://www.research.digital.com/SRC/WebL/.

[62] WinBench. http://www.zdnet.com/zdbop/winbench/winbench.html, Ziff-Davis.

[63] Wollman, J. D., *Hbench:Proxy – A Realistic Proxy Server Benchmark*, Senior thesis, Harvard University, 1999.

[64] Zhang, X., Wang, Z., Gloy, N., Chen, J. B., and Smith, M. D., "System Support for Automatic Profiling and Optimization." In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 15-26, St. Malo, France, October 5-8, 1997.

[65] Zhang, X., and Seltzer, M., "HBench:Java – An Application-Specific Benchmarking Framework for Evaluating Java Virtual Machines." In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 62-70, San Francisco, CA, June 3-4, 2000.

[66] Zhang, X., and Seltzer, M., "HBench:JGC – An Application-Specific Benchmark Suite for Evaluating JVM Garbage Collector Performance." In *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '01)*, pages 47-59, San Antonio, Texas, January 29 – February 2, 2001.

[67] Zorn, B., "The Measured Cost of Conservative Garbage Collection." *Software–Practice and Experience*, 23(7), pages 733-756, July 1993.