HBench:Java: An Application-Specific Benchmarking Framework for Java Virtual Machines

Xiaolan Zhang Margo Seltzer
Division of Engineering and Applied Sciences
Harvard University
33 Oxford Street
Cambridge, MA 02138, USA
617-495-3311, 617-496-5663
{cxzhang, margo}@eecs.harvard.edu

ABSTRACT

Java applications represent a broad class of programs, ranging from programs running on embedded products to high-performance server applications. Standard Java benchmarks ignore this fact and assume a fixed workload. When an actual application's behavior differs from that included in a standard benchmark, the benchmark results are useless, if not misleading. In this paper, we present HBench:Java, an application-specific benchmarking framework, based on the concept that a system's performance must be measured in the context of the application of interest. HBench:Java employs a methodology that uses vectors to characterize the application and the underlying JVM and carefully combines the two vectors to form a single metric that reflects a specific application's performance on a particular JVM such that the performance of multiple JVMs can be realistically compared. Our performance results demonstrate HBench:Java's superiority over traditional benchmarking approaches in predicting real application performance and its ability to pinpoint performance problems.

Keywords

Java performance, benchmarking.

1. INTRODUCTION

In recent years, the Java programming language has enjoyed increasing popularity and there has been a proliferation of Java Virtual Machine (JVM) implementations. This poses a question for end users: which JVM should they choose to run their applications? There have been many attempts to evaluate different JVM implementations. Unfortunately, these approaches share a common drawback: they assume a fixed set of workloads and ignore the application's performance concerns. Java applications represent a diverse set of programs, ranging from those running on embedded products such as PDAs, to applets running in browser environments, to scientific computing applications, and recently to server applications, which have traditionally been the stronghold of system languages such as C and C++. Often the actual application under test differs enough from any standard benchmark that the results from traditional benchmarks are useless and sometimes even misleading. Moreover, since the workloads are fixed, traditional benchmarks encourage vendors to over-optimize their JVM implementations to achieve good results on the benchmarks. This may potentially hurt the performance of real applications. Such incidents have already been reported in the area of OS benchmarking, where graphics card vendors employ a hack, which can severely hamper the performance of other devices, to improve their results in standard benchmarks [8].

We believe that the goals of benchmarking in general should be threefold:

1. To compare the performance of systems and to reason about why applications run faster on one system than on another. Not only should benchmarks produce meaningful results, they should also provide a reasonable explanation for the performance differences.
2. To guide performance optimizations. Benchmarks should reveal performance bottlenecks or limitations of the underlying system in the context of a particular application, and thus help system implementers improve the system in a way that will benefit the application of interest.
3. To predict an application's performance on non-existent platforms. Benchmarks should help answer "what if" questions and provide users with a reasonable estimate of the application's performance when some components of the underlying system change, or when the behavior of the application changes.

In this paper, we present HBench:Java, part of a more general application-specific benchmarking framework called HBench designed to realize the above goals.

The rest of the paper is organized as follows. Section 2 gives an overview of some of the most popular standard Java benchmarks. Section 3 describes the design of HBench:Java, and Section 4 describes our prototype implementation of HBench:Java in detail. Section 5 presents experimental results.

Section 6 describes some related work. Section 7 discusses some unresolved issues and Section 8 concludes.

2. JAVA BENCHMARKS

Traditional Java benchmarks can be classified into the following three categories:

1. Microbenchmarks. CaffeineMark [3] is a typical example, in which a set of JVM primitive operations such as method invocation, arithmetic and graphics operations, and short sequences of code (kernels) that solve small and well-defined problems, are measured, and the mean (typically geometric mean) of the individual times (or scores as a function of the time) is reported. Microbenchmarks are useful in comparing the low-level operations of JVMs, but it is difficult to relate them to actual application performance in a quantitative way.
2. Macrobenchmarks that contain one or more medium-scale to large-scale Java applications. Examples include the SPECJVM98 suite [15], which includes a set of programs similar to those found in the SPECCPU suite, and VolanoMark from Volano LLC, which is based on the company's VolanoChat™ server. VolanoMark focuses on a JVM's ability to handle "long-lasting network connections and threads" [16].
3. Combinations of the above. The JavaGrande benchmark [2][9] is an example of this type. 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 such as arithmetic operations, mathematical functions, and exception handling. The second section consists of kernels, each of which contains a type of computation likely to appear in large scientific programs. The final section includes realistic applications, such as a financial simulation based on Monte Carlo techniques. This hybrid approach of combining microbenchmarking and macrobenchmarking 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.

The common drawback with the above approaches is that Java applications are so diverse that it is difficult, if not impossible, to find a set of workloads that are representative of the applications in which end users are interested, 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 JVM is the best for the application of interest. In comparison, HBench:Java is a general benchmarking framework that can be applied to any specific workload.

3. HBENCH:JAVA DESIGN

3.1 Overview

HBench:Java is based on the vector-based methodology of the HBench framework [14]. The principle behind the vector-based methodology is the observation that a system's performance is determined by the performance of the individual primitive operations that it supports, and that an application's performance is 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 a JVM, with each entry v_i representing the performance of a primitive operation of the JVM. We call this vector V_s a *system vector*, and it is obtained by running a set of microbenchmarks.

A key feature of HBench:Java is that it incorporates characteristics of the application into the benchmarking process. This is achieved using an *application vector*, $V_A = (u_1, u_2, ..., u_n)$, with each element u_i representing the number of times that the corresponding i^{th} primitive operation was performed. Intuitively, the application vector indicates how much demand the application places on the underlying JVM and is obtained through profiling. The dot product of the two vectors produces the predicted running time of the application on a given JVM.

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. For example, on some systems, TCP connect times grow non-linearly with the number of connections. Rather than modeling the non-linearity explicitly, we provide three or four points in the system vector that correspond to differing orders of magnitude for the number of connections.

HBench:Java addresses the benchmarking goals outlined in Section 1 in the following ways:

1. The system vector and the application vector provide an effective way to study and explain performance differences between different JVMs.
2. The application vector indicates which primitive operations are important, and the system vector reveals which primitive operations are performance bottlenecks. System implementers can use this information to improve primitive operations that are significant for the application. At the same time, application programmers can use this information to optimize the application by reducing the number of calls to expensive primitive operations.
3. One can predict the performance of the application on a given JVM without actually running the application on it, as long as the system vector is available[1]. One might also answer "what if" questions such as "What if this primitive takes twice as long?" by modifying the appropriate system and application vector entries.

[1] HBench:Java will work best with support from JVM vendors who supply the system vectors for their JVM products.

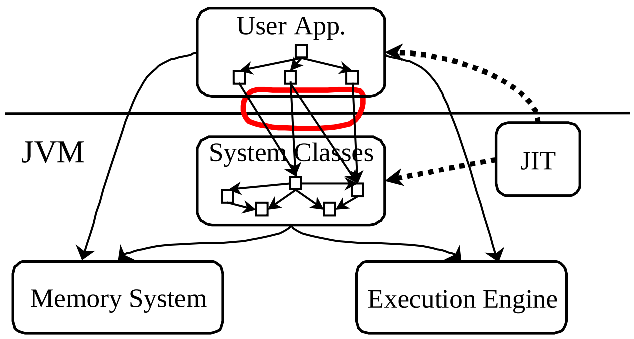

Figure 1. Schematic view of a JVM.

3.2 Identifying Primitive Operations

A JVM is a complicated piece of software. Figure 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). A JVM includes a memory management system that automatically manages the heap for the application. The execution engine is responsible for bytecode interpretation, class loading, exception handling, thread scheduling and context switches, the native method interface, and synchronization. The JVM implementation is further complicated by the *JIT (Just In Time)* component, which compiles Java bytecode on the fly into native machine code.

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

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

Figure 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 2(b). Corresponding native code sequences

3.2.1 First Attempt

In order to create a system vector for a JVM, we need to decompose this complexity into a set of primitive operations. One set of candidates is the JVM's assembly instructions, i.e., bytecodes. 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 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 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.

3.2.2 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, capturing the four major components of a JVM implementation as depicted in Figure 1, namely, system classes; memory management; execution engine; and JIT compiler. The following subsections describe each component and its primitive operations in details.

3.2.2.1 System Classes

Identifying the primitive operations for the system classes component is straightforward — we use method invocations to the system classes, published in the standard Java API specification, as primitive operations.

3.2.2.2 Memory Management

We consider two primitive operations of the memory management component, namely, object allocation and dead object reclamation[2].

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

1. size of allocation,
2. status of the heap, such as number of free blocks and their sizes.

We can represent this cost with a function C_{alloc}*(heap_status, allocation_size)*. Depending on the memory management algorithm, C_{alloc} takes on different forms. In the case of copying garbage collectors, the free space is a contiguous area, and allocation can be implemented with a simple pointer advancement. Therefore, in this particular case C_{alloc} is just a constant function. In the case of non-copying collectors, such as mark and sweep collector, the allocation time depends on the status of the free-block lists maintained by the collector. If we characterize the heap status with simple statistical measures,

[2] More details in this subject will be presented in a forthcoming paper.

such as a normal distribution with certain mean and standard deviation, or a uniform distribution with a certain 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.

An interesting fact with 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 the dead object to the available space. In most cases, this involves updating certain 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 object, 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:

$$T_{GC} = C_{fixed}(h) + \sum_{s} C_{live}(s) * N_l(s) + C_{dead} * \sum_{s} * N_d(s)$$

3.2.2.3 Execution Engine

Primitive operations of the execution engine include bytecode interpretation, exception handling, context switching, synchronization operations, etc.

3.2.2.4 JIT Compiler

Performance of the JIT compiler can be measured using 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 N, and S, where each entry in N 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 i^{th} element of S. The quality of JITted-code is harder to quantify, and is a subject of ongoing research.

4. HBENCH:JAVA IMPLEMENTATION

The HBench:Java prototype implementation currently includes only the system classes component, as highlighted by the circle in Figure 1. 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.

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.

4.1 Profiler

The profiler is based on JDK's Java Virtual Machine Profiling Interface (JVMPI) [7]. 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 **Error! Reference source not found.**. 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 more complete custom profiler that incorporates the garbage collector (GC) and the JVM execution engine and that is able to directly produce an application vector is currently under development.

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) [6]. 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.

4.2 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 5.

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 2*n* times the timer resolution (so that the benchmark doesn't run for an unnecessarily long time). For the experiments reported in this paper, 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. Our current prototype implementation supports this simple case of linear dependency on a single argument, and we found it sufficient for the sample applications we 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 i^{th} 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.

Figure 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.

4.3 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 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 [5] 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*.
- To measure the per-bytecode compiler/optimize overhead, the java.lang.Compiler class should be augmented with APIs for compiling and optimizing methods.

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 3. Sample microbenchmark results.

Table 1. Java Virtual Machines tested.

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	UltraSparc 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 2. Java applications used in the experiments.

Application	Description	Input Data
WebL	A scripting language designed specifically for processing documents retrieved from the web [17].	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 [4].	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 [11].	The synthetic proxy provided by the Mercator kit that generates web documents on the fly instead of retrieving them from the Internet.

5. EXPERIMENTAL RESULTS

5.1 Experimental Setup

We ran our experiments on a variety of Java Virtual Machines. Table 1 shows the list of JVMs tested and their configurations.

Three non-trivial Java applications (Table 2) were used to evaluate HBench:Java. First, we ran the applications with profiling turned on and derived application vectors from the collected profiles. Next we ran the HBench:Java microbenchmarks on the JVMs listed in Table 1 and obtained 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 our initial goal is to correctly predict the ratios of execution times of the applications on different JVM platforms, we use normalized speed in reporting experimental results. This also allows us to compare HBench:Java with conventional benchmarking approaches such as SPECJVM98 that report results in the form of ratios.

5.2 Results

Figure 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 3. Also shown in Table 3 are the measured performance on the five Java Virtual Machine tested. The corresponding application vector is (80, 121, 32768). It's interesting to note that the SPECJVM98 score of JDK1.2.2 on the PentiumPro NT machine is higher than that on the SparcStation. However, WebL runs close to three times faster than on the SparcStation. HBench:Java's system vector reveals the problem. Class loading is twice as fast for the SparcStation 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 SparcStation. Besides explaining performance differences, the predicted ratios of execution speeds are within a small margin of the real execution speed ratios.

Figure 5 shows the results for Cloudscape, a database management system. We did not report the result for the Sun JDK1.2.2 classic version on the SparcStation 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 large amount of time in class loading.

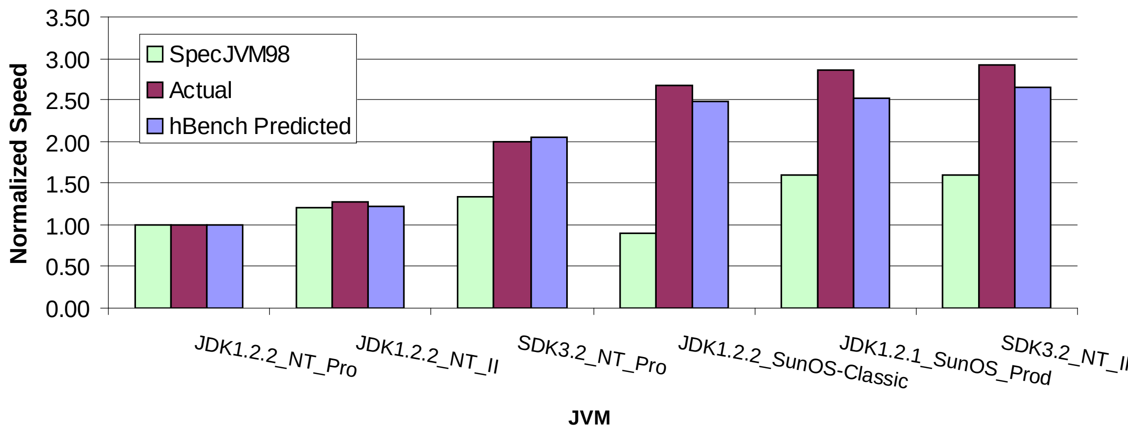

Figure 4. Normalized running speeds for WebL.

Table 3. Important primitive operations for WebL.

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

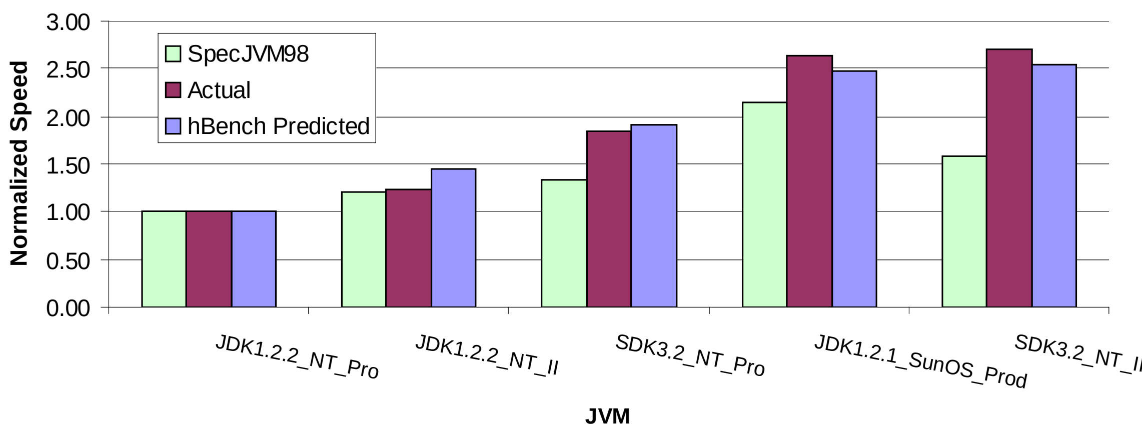

Figure 5. Normalized running speeds for Cloudscape.

Figure 6 shows the results for Mercator, the web crawler. We ran the proxy server and the web crawler on two different machines connected with a 100Mb Ethernet switch, isolated from the outside network. The 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. We only collected results for a limited number of JVMs due to the

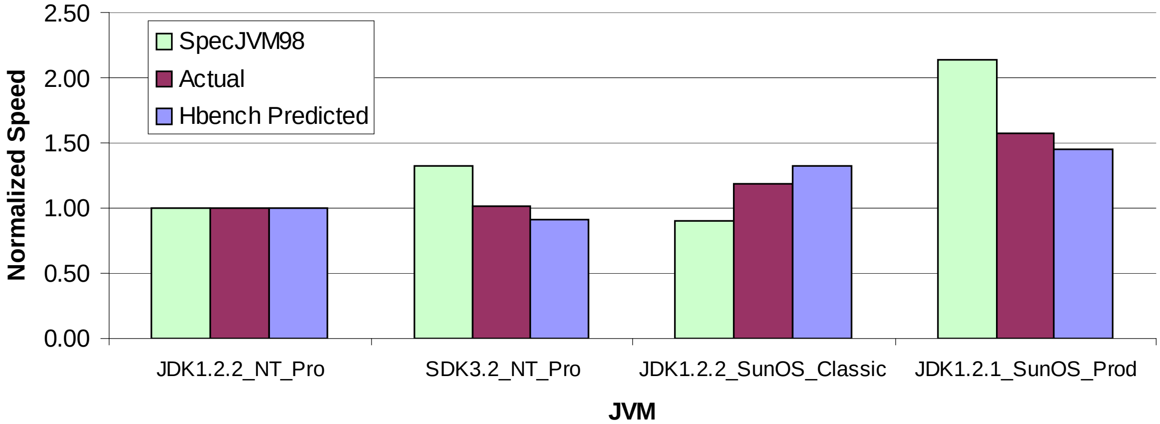

Figure 6. Normalized running speeds for Mercator.

Table 4. Important primitive operations for Mercator.

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

difficulty of setting up the machines in an isolated network[3]. 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. Table 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).

To understand why SPEC performs poorly, we examined the time breakdown for user versus system classes. Tables 5 and 6 show the percentage of time spent in system classes for SPEC 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

[3] We have an agreement with Compaq that requires experiments concerning Mercator to be run in an isolated (disconnected) network environment.

Table 5. Time breakdown for SPECJVM programs.

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 6. Time breakdown for sample applications.

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

data show, the SPEC programs spend most of the time in user classes. Therefore, they are poor predictors for applications that spend a lot of time in system classes. 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.

In theory we can use HBench:Java to predict the running time of SPEC programs. However, since SPEC programs spend little time in system classes, the few system classes serve as poor sample data for measuring JIT quality, resulting in large error rate. Therefore, for SPEC-like applications, more sophisticated techniques to measure JIT quality are needed.

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. We expect the accuracy of HBench:Java to improve as the system vector is completed.

6. RELATED WORK

The HBench:Java approach is similar to the *abstract machine model* [10], where the underlying system is viewed as an abstract Fortran machine, and each program is decomposed into a collection of Fortran abstract operations called *AbOps*. 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 [13]. As hardware becomes more sophisticated, the accuracy achievable with this technique tends to decrease. This is the key reason we did not use bytecodes as primitive operations.

Brown [1] used the vector-based approach of HBench to evaluate operating systems. They demonstrated that it effectively predicts the performance of the Apache web server on different platforms. The primitive operations in this case are system calls, and the application vector is essentially the system call trace.

7. DISCUSSION AND FUTURE WORK

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. Our goal is to keep the number of microbenchmarks for the system class method calls under 200.

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.

Our short-term goal is to implement a complete set of system class microbenchmarks for HBench:Java and to test it on more JVM varieties and commercial applications. In the long run, we will implement other parts of the system vector, including components representing the memory system and the execution engine.

8. CONCLUSION

HBench:Java is a vector-based, application-specific benchmarking framework for JVMs. Our 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.

9. ACKNOWLEDGMENTS

We wish to thank Allan Heydon and Marc Najork at Compaq Systems Research Center for providing us with the Mercator kit, and Suiling Ku for her assistance with the Cloudscape software. We also thank the creators of the WebL scripting language. Special thanks go to Lance Berc from Compaq Systems Research Center for his suggestions of Java applications for measurement. We are grateful to the anonymous reviewers for their insightful comments and suggestions on improving this paper, and to David Sullivan, Keith Smith and Kostas Magoutis for proofreading the draft. David Sullivan implemented the first version of the HBench:Java microbenchmarks.

10. REFERENCES

[1] 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.

[2] 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.

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

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

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

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

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] Seltzer, M., Krinsky, D., Smith, K., and Zhang X. The Case for Application-Specific Benchmarking. In *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems (HotOS VII)*, pages 102-107, Rio Rico, AZ, March 29-30, 1999.

[15] SPECJVM98. http://www.spec.org/osg/jvm98/.

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

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