RC 21858 (98373) 20 October 2000 **Computer Science**

IBM Research Report

A Characterization of Java Server Benchmarks

Michael Kalantar
IBM Research Division
Thomas J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
kalantar@us.ibm.com

Sandra Johnson Baylor
IBM Silicon Valley Laboratory
55 Bailey Avenue
San Jose, CA 95141
sandrajb@us.ibm.com

Xin Li
Department of Electrical and Computer Engineering
Northwestern University
Evanston, Illinois 60208
lix@ece.nwu.edu

Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

Abstract

With its promise of "write once, run anywhere" development, the Java platform has the potential to revolutionize computing,. However, in order to realize this potential, Java applications, or those using Java extensions, frameworks, or components, must exhibit a competitive level of performance. In addition, in order to optimize the performance of Java applications, there must be some level of understanding of the generic trends that exist in these applications which are amenable to performance improvements. This paper presents preliminary results of a study conducted to characterize the workload of Java server benchmarks. The study is motivated by a desire to understand some of the general behaviors of these types of programs in order to develop more effective compiler optimizations, memory subsystem designs, garbage collectors, etc., resulting in better performance. The results we present characterize the frequency of various method invocations exhibited by these benchmarks. They show that the most frequently invoked methods are from the Java classes, particularly, `java.lang`, `java.util` *and* `java.io`*. Also, within java.lang, the most frequently invoked methods are from the* `String` *class.*

1. Introduction

The Java platform has the potential to revolutionize computing, with its promise of "write once, run anywhere" development. This will enable application developers to develop an application only once, therefore minimizing or eliminating its porting to multiple platforms. In order to realize this potential, Java solutions must exhibit satisfactory levels of performance. One method that can be used to optimize performance is to understand the general behavior of a diverse set of Java server applications and then target specific optimizations, based upon this behavior.

While there has been some activity on the performance evaluation of Java client applications, it is our belief that Java server applications will dominate in the future. There are issues that are specific to server applications, such as multi-threaded object allocation and concurrent resource access from hundreds or thousands of threads. Also, Java server extensions and servlet support are important components of Java server applications. It is important that we understand the

workload characteristics of Java server applications, in order to effectively and efficiently optimize their performance.

As a preliminary step to characterizing Java server applications, we present the results of a study conducted to characterize, at a very high level, the workload of several Java server benchmarks. These benchmarks are thought to be representative of Java server applications and have been used extensively to guide the development and optimization of Java virtual machines and to evaluate a variety of platforms' effectiveness at executing Java server code. The objective of the study is to understand some of the general behaviors of these types of programs in order to develop more effective compiler optimizations, memory subsystem designs, garbage collectors, etc., resulting in better performance.

The next section describes the Java server benchmarks evaluated in this study. Section 3 presents the methodology used to collect the workload characteristics. Section 4 presents the results and Section 5 discusses the related work. Finally, Section 6 summarizes the work presented and suggests directions for further study.

2. Benchmarks

A diverse collection of server benchmarks is used in this study. They include a business object benchmark for Java (jBOB), a portable business object benchmark (pBOB), and a web server benchmark (JWeb2). These benchmarks are described and discussed in detail in [BAYLR00]; however, we include general descriptions of them here for completeness.

The jBOB benchmark is a simple transactional server application for a typical electronic business scenario. The benchmark's workload is a mixture of read-only and update intensive transactions that simulate the activities found in complex online transaction processing (OLTP) application environments. The benchmark is representative of applications exercising a breadth of system components characterized by concurrent transactions, on-line and deferred transaction execution modes, moderate system and application execution times, transaction integrity, non-uniform distribution of data access though primary

and secondary keys, and databases consisting of many tables with a wide variety of sizes, attributes and relationships. Application drivers are used in jBOB to generate and submit transaction results from clients to the server; however, its objective is to quantify the performance of the server-side application. Therefore, the performance of the client and the network is not included in the quantified metric.

The jBOB implementation contains three logical tiers. The first tier, the clients, generates data access requests modeled after specific transaction types. The second tier, a thin layer of Java code, acts as a front end to the third tier database server. The benchmark exercises the JDBC, or the Structured Query Language (SQL) embedded in Java (referred to as SQLJ), interface, Java communication using Java sockets, core JVM and class level synchronization, data conversion, multi-threading, object serialization, object creation, and garbage collection. The benchmark reports the performance metrics of throughput and transaction response time.

The pBOB benchmark simulates a typical transaction-oriented workload. It was designed to analyze the processor/memory subsystem, and associated software, for typical server-based business applications. This is in contrast to jBOB, which is used to study the complete server configuration. To achieve a pure Java implementation, pBOB replaces third-tier database tables with in-core Java classes and records. There is no persistence or transaction control; instead, all transaction domain instances live only for the duration of the JVM instance. pBOB does not perform terminal emulation or communication. Instead, Java threads represent terminals in pBOB, where each thread independently generates random input before calling transaction specific logic. pBOB has no external dependencies and is portable to any compliant JVM.

The pBOB workload is controlled by varying the number of warehouses, the object population per warehouse, the number of threads (representing terminals) per warehouse, keying and think times,

number of order lines per order, and whether initial and transaction result screens are displayed. pBOB also supports a "steady state" mode where the creation of new objects is balanced by the removal of old objects. This allows long duration tests to run in finite heap space.

JWeb2 is a dynamic web-serving benchmark that characterizes the behavior of a simple e-commerce application whose clients are HTTP-based web browsers. JWeb2 is derived from the SpecWeb99 [SPEC99] benchmark, in which simulated clients access a series of URL-specified HTML pages. JWeb2 has enhanced this by simulating an e-commerce application and supplying implementations of this application as a servlet. JWeb2 exercises web server based Java program execution, multiple threads, session management, file I/O, and a server's capability to handle high HTTP traffic and memory access bandwidth.

JWeb2 is designed to return 50% dynamic content and 50% static content, based on the assumption that each dynamic page will typically refer to at least one image. Second, the sizes of the static files and the results of related dynamic requests follow a distribution similar to that expected on a typical web server. In particular, distributions were chosen to match those used by the SPECWeb99 benchmark [SPEC99]. However, in our experiments for this study, we reconfigured the benchmark to return 100% dynamic content.

A typical e-commerce web site uses cookies or URL rewriting to maintain session state information about each user. If passwords or cookies are used, the web server may rely on a database of individual preference. Typical web server application activities include text manipulation, page development, data lookup (either via a database or file access) and table generation. Communication with clients may or may not be encrypted. Most interactions with an e-commerce web site involve a number of steps. The user first logs on, creating a new session. Second, the user may search the web site, access information,

or engage in a transaction. Finally, the user logs off. Each of these activities may be carried out by a different dynamic program.

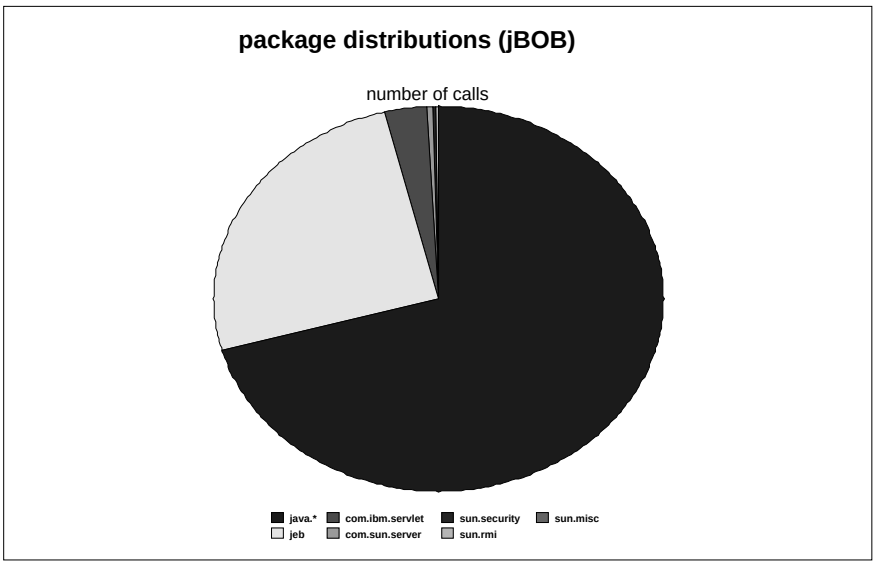
The JWeb2 benchmark abstracts the above activities into a single Java servlet. This servlet first manages session information: it uses a cookie to identify the session, creating a new one if necessary. Second, the session state, a record of all previous accesses during the current session and the number of rows in the reply, is read and the contents output to a newly created HTML page. Third, the parameters to the dynamic request are used to access a file. The file is read and is used to generate a summary table. Finally, if the session is to be terminated, all session state is invalidated. Otherwise, information about the current request is added to the session.

3. Collection Methodology

We ran our experiments in a Windows NT 4.0 (SP4 4) environment on a 4-way Pentium Pro 200MHz machine with 2 GB of main memory. Each of benchmarks detailed above have different requirements. pBOB, is entirely self-contained; it runs within a single java virtual machine. Both clients and server are bundled together. The other two are servlet based. Each is accessed from remote clients through a web server (Internet Information Server (IIS), version 4) which passes

Figure 1. Package Method Invocation Frequency for jBOB.

servlet requests to a web application server (WebSphere Application Server (WSAS), version 2.02). JWeb2 uses the file system, while jBOB uses a database (DB2, version 5.2). JWeb clients are run on AIX machines on the same local area network (16Mbit/s token ring) and jBOB clients are run on NT workstations also on the same local area network. We found that successfully running our experiments, despite the use of Java, remained a challenge. In part this is a

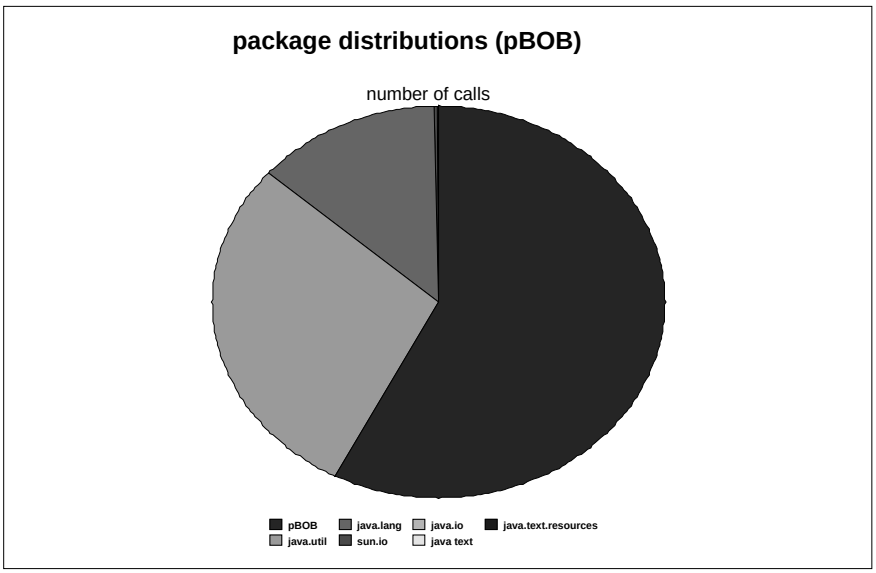


consequence of the fact that most measurement software requires a specific virtual machine. These virtual machines may not be compatible with the middleware used (in our case, the version of WSAS we used required JDK 1.1.7).

To collect the high-level data we were initially interested in, we considered using the following tools: JINSIGHT [JIN99], Jprobe [JPRBE99], OptimizeIt [OPT99], and VTune [VTUNE99]. These tools are similar in that they are interactive tools designed for program performance, heap analysis and optimization. Each requires a tool specific virtual machine which produces the

Figure 2. Package Method Invocation Frequency for pBOB.

necessary profiling information. Furthermore, JINSIGHT, JProbe and OptimizeIt all interpret byte code: they can not run JIT'ed code. As a consequence, we found that our benchmarks run up to 30 times slower when using these tools. Of these tools, JINSIGHT and OptimizeIt are strictly interactive tools; they do not provide a means for extracting data for later analysis. We selected JProbe for our study because it proved to be the most stable and we were able to collect profiling

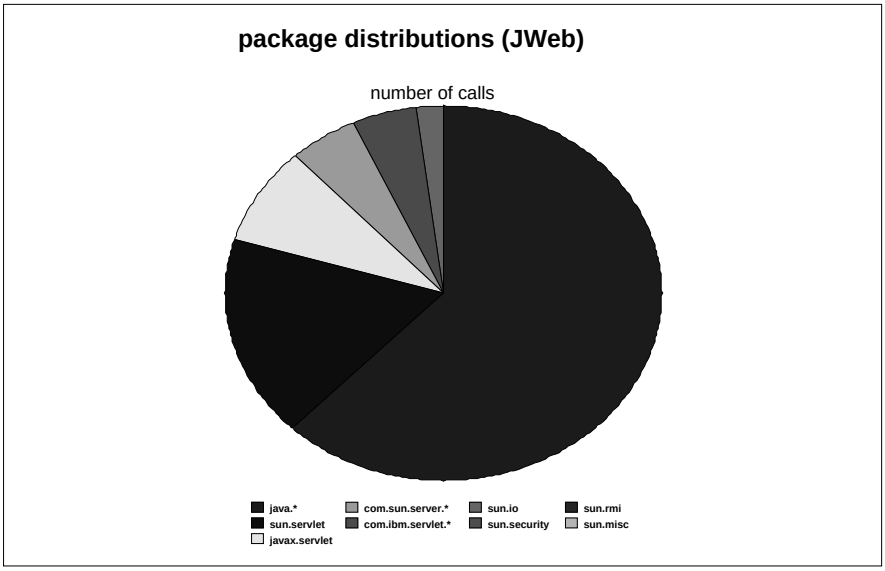


data. In the case of jBOB, the slowdown caused by using JProbe eventually results in client time-outs and an assumption of transaction failures. We were able to run VolanoMark [VOL99] using JProbe, but were unable to collect profiling data for this benchmark.

JProbe is capable of collecting two types of data: performance and heap data. This data can be displayed interactively as it is collected or periodically collected in snapshots. Data can be collected on a method by method or a line by line basis. We collected our data on a method by method basis. For each method five types data are collected; the number of calls made, the

Figure 3. Package Method Invocation Frequency for jWeb.

cumulative execution time, the method execution time, the cumulative number of objects allocated and the method number of objects collected. The cumulative time (number of objects allocated) represents the time spent (number of objects allocated) in the method and its sub-tree in the call graph. A method time (number of objects allocated) represents the time (number of objects allocated) in the method itself not including the time spent in the sub tree of the call



graph. Times can be collected as either CPU time or elapsed time. Further, data collection can be filtered to either include or ignore native methods, and can be filtered to either include or ignore the execution of an application server (JProbe is designed to be aware of many application servers, including WebSphere). In all cases, we collected data which included the execution of the native methods and the application server.

JProbe collects further data that enables one to interactively construct method call graphs instrumented by frequency of path execution. We have not yet been able to use this information. We believe a characterization of call graph behavior would be helpful in compiler construction [FINK00]. Rather, we analyze the raw numbers without call graph information.

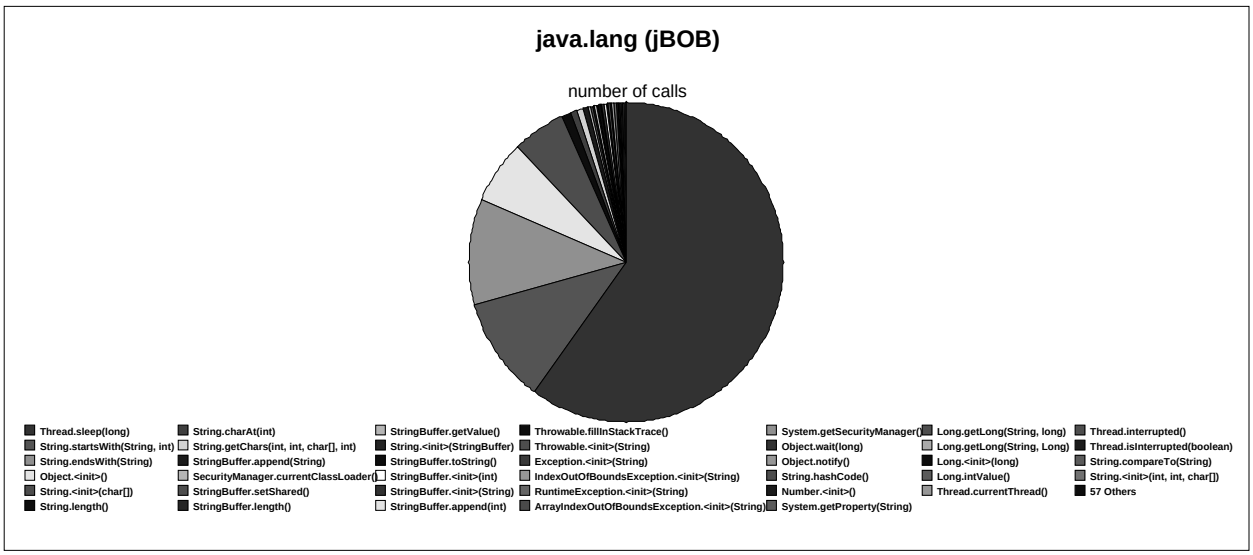


Figure 4. `Java.lang` Method Invocation Frequencies for jBob.

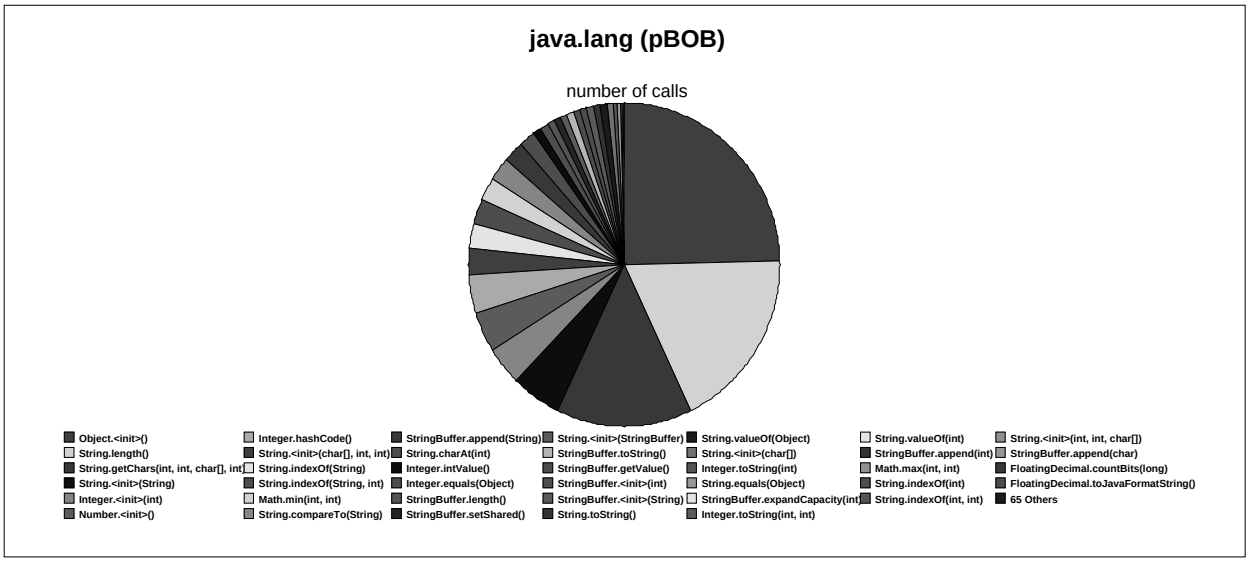


Figure 5. `Java.lang` Method Invocation Frequencies for pBOB.

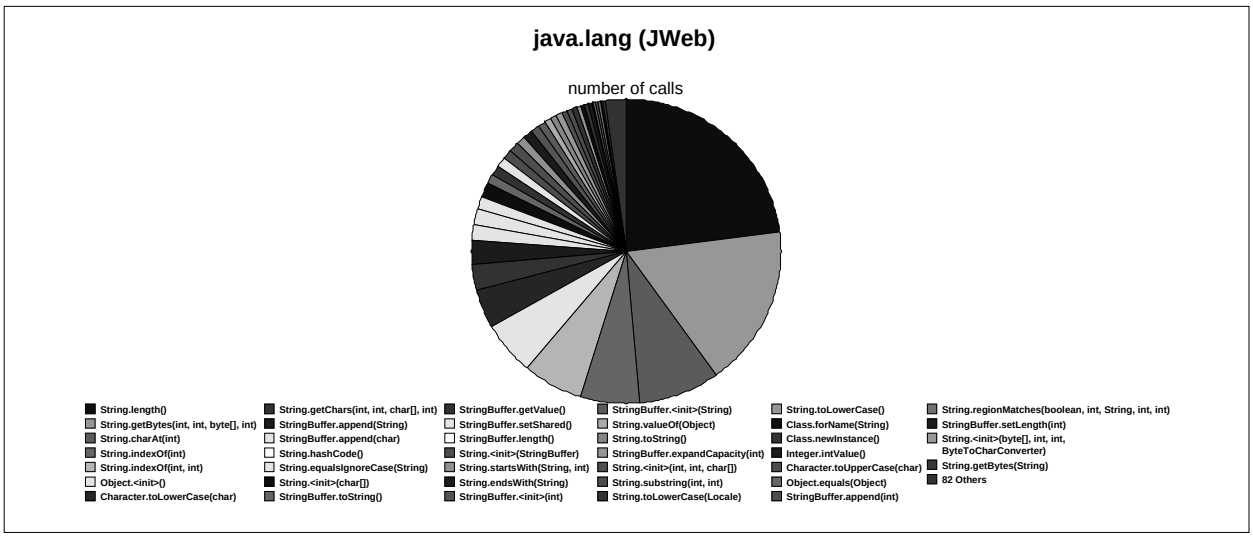


Figure 6. `Java.lang` Method Invocation Frequencies for jWeb.

4. Results

Illustrated in Figures 1-3 are the Java package method invocation distributions for jBOB, pBOB and JWeb2, respectively. The figures show the Java classes exhibiting the largest fraction of method invocations for jBOB and JWeb. The next highest number of method invocations are for the jeb classes for jBOB and servlets for JWeb2. This reflects the fact that JWeb2 is implemented as only a few methods. Further, it makes heavy use of the servlet classes to extract session information, cookies and parameters. jBOB requests, on the other hand, have few parameters, do not use sessions or cookies, and require more internal processing. In contrast, for pBOB, the largest fraction of method invocations are from the pBOB class, followed by a number of Java classes. Methods in the pBOB benchmark are called so frequently due to the inclusion of many pieces into the benchmark: the server processing, an in-core database and client processing. If the pBOB classes are ignored, methods from the Java class library represent

nearly 70% of all method calls. In general these results show the Java classes exhibiting the largest fraction of method invocations.

We took a closer look at these Java classes and observed that the overwhelming majority are found in `java.lang`, `java.util` and `java.io` for all benchmarks evaluated. We then examined the method invocation frequencies for each of these classes. Presented in Figures 4-6 are the method invocation distributions for `java.lang`. In these figures, each slice of the pie represent the percentage of `java.lang` method invocations that are attributed to the specific sub-class (slice).

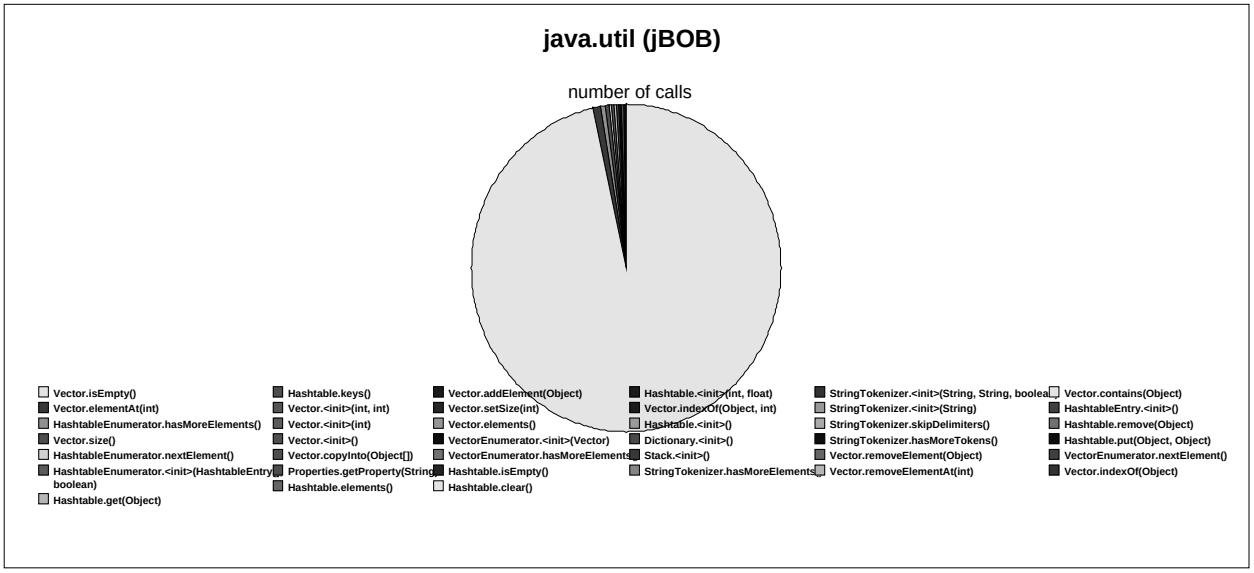


Figure 7. `Java.util` Method Invocation Frequencies for jBOB.

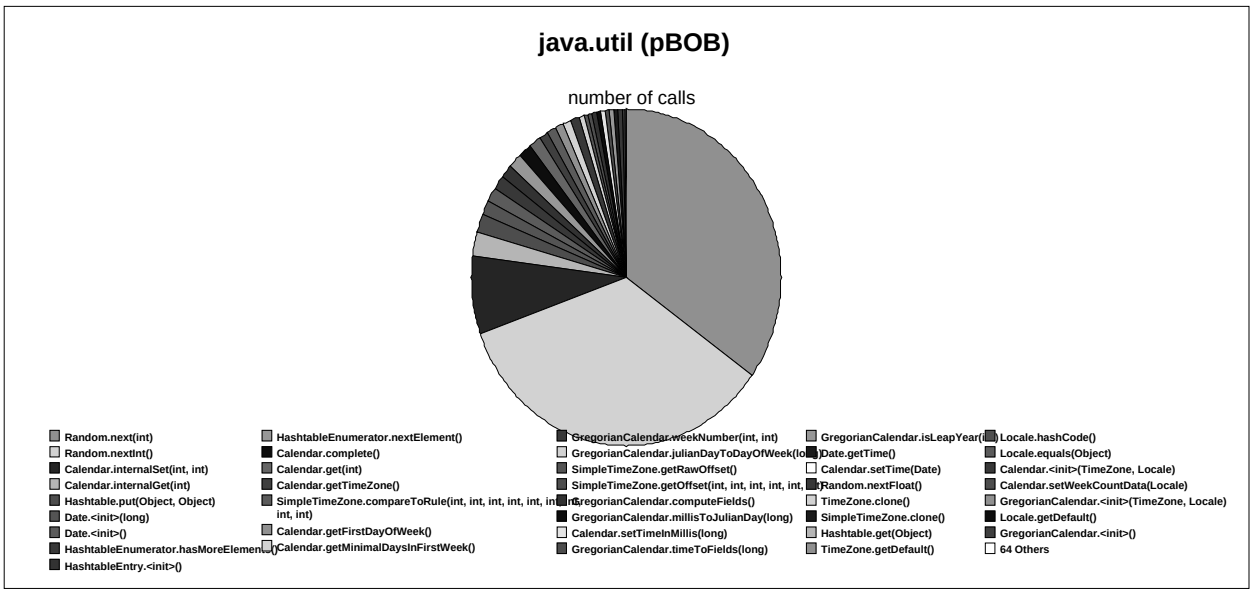


Figure 8. `Java.util` Method Invocation Frequencies for pBOB.

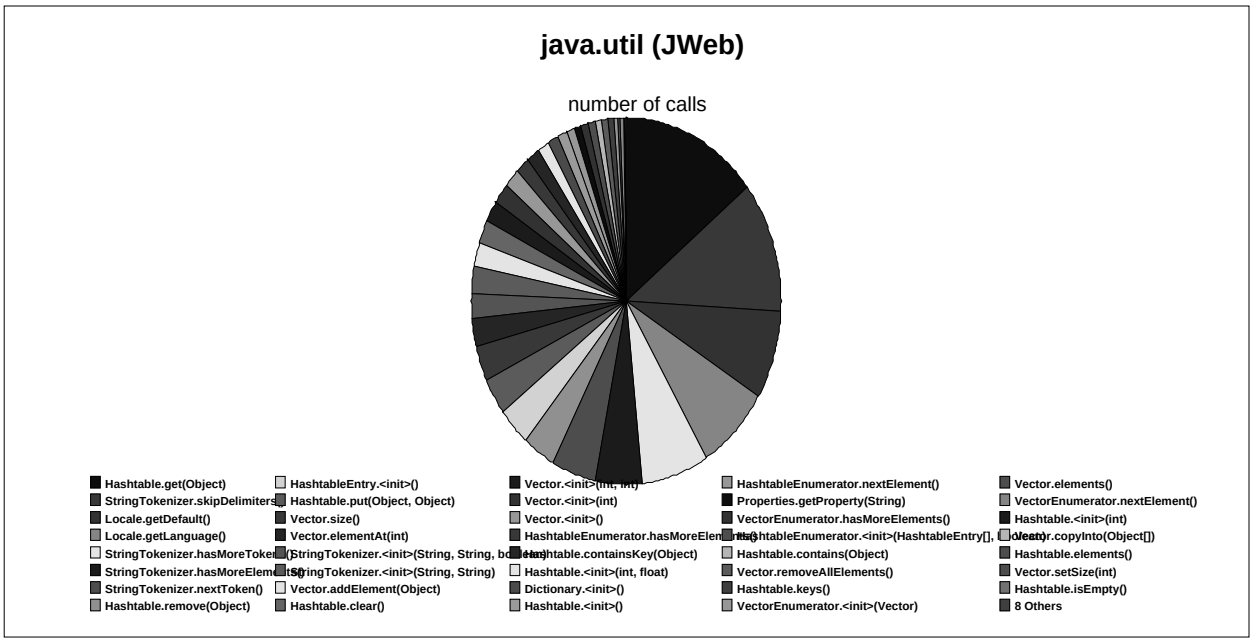


Figure 9. `Java.util` Method Invocation Frequencies for jWeb.

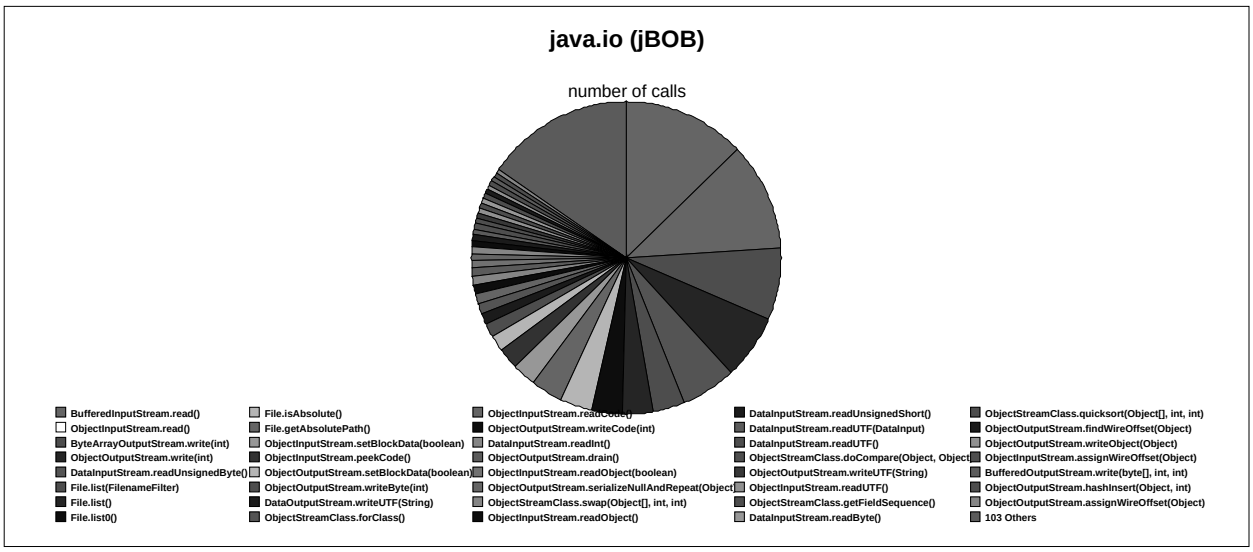


Figure 10. `Java.io` Method Invocation Frequencies for jBOB.

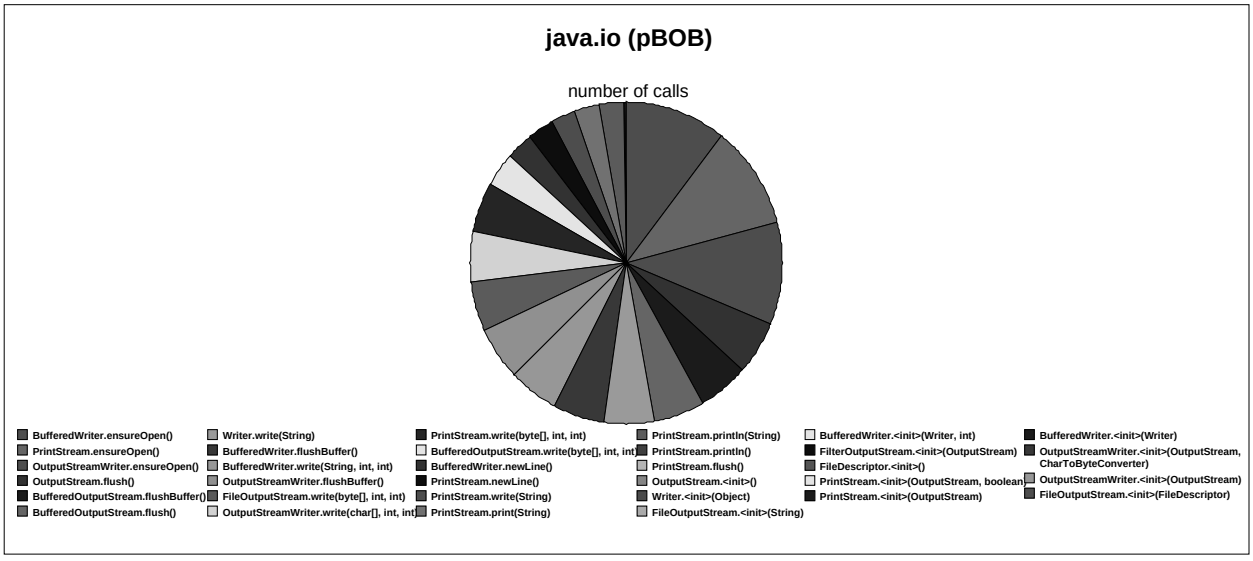


Figure 11. `Java.io` Method Invocation Frequencies for pBOB.

These figures show a large fraction of the `java.lang` methods invoked are from the `String` class. Also, for jBOB, the Thread.sleep method is invoked the majority of the time and for pBOB, Object.init is invoked frequently, as many objects are allocated as a part of the in-core database calculations.

Presented in Figures 7-9 are the `java.util` method invocation distributions for the benchmarks evaluated. Also, Figure 10-12 illustrates the `java.io` method invocation distributions for these benchmarks. These results show that only a few methods dominate in the number of times they are invoked. They include Vector.IsEmpty for jBOB, Random.next and Random.nextInt for pBOB, and Hashtable.get and StringTokenizer.skipDelimiters for JWeb2 for `java.util`. In the case of pBOB, the high number of calls to Random methods is a side effect of the in-core database simulation. In JWeb2, the calls to frequent methods are related to session state information and user request parsing. Such costs would, we believe, be typical for a servlet taking full advantage of such facilities. For `java.io`, they include BufferedInputStream.read for jBOB and JWeb2 and several BufferedWrite and OutputStream methods for pBOB. In the cases of jBOB and JWeb2, input from a database and from the file system are major components of the benchmarks. Because pBOB is self contained, it does not include input methods. Rather, its output methods are weighted higher.

5. Related Work

Several studies have evaluated the workload characteristics of Java applications. Conte, *et. al.* [CONTE98], characterized Java application and applet workloads as it relates to the reuse and sharing of Java codes at the program, class and method levels. They used the SPECjvm98 benchmarks and 3 extensive searches of the Internet. Their objective was to determine potential compiler optimizations, based upon any observed general trends. They found sufficient code reuse to warrant further research on optimizations, based upon the results.

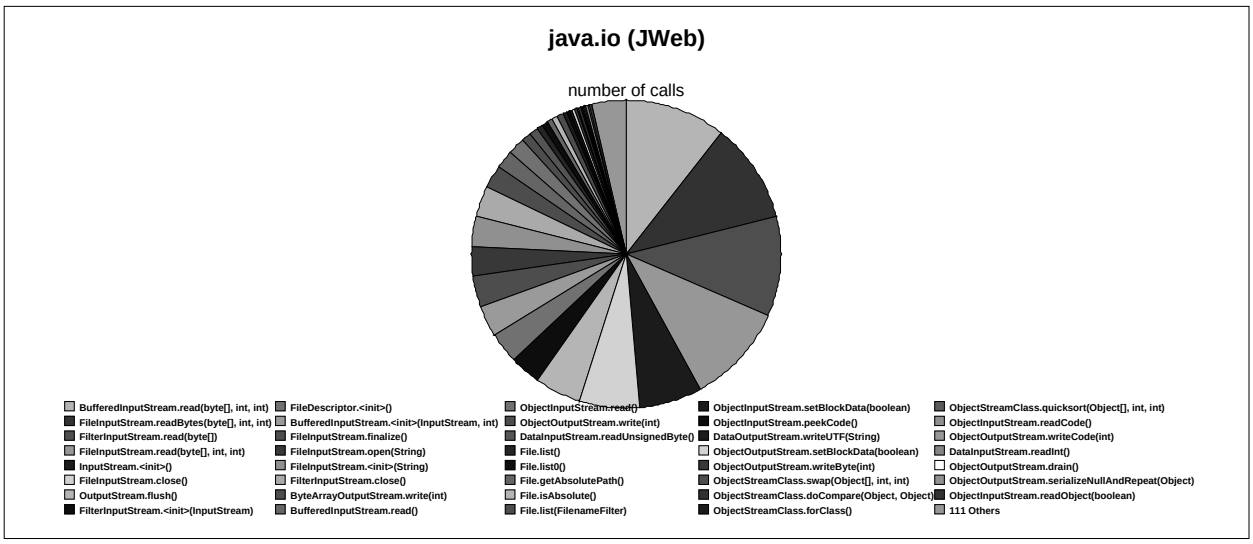


Figure 12. `Java.io` Method Invocation Frequencies for jWeb.

Chow, *et. al.* [CHOW98] examined the branching characteristics of about 1000 instruction traces of both Java and non-Java workloads. Their objective was to do a functional examination and to apply any knowledge gained to the design of current and future computer systems. They used multivariate analysis and determined that Java workloads tend to have a greater number of indirect branches and no significant differences between direct and conditional branching characteristics.

Dieckmann and Holzles [DIECK99] conducted a study that focused on the allocation of Java applications to identify methods to better identify and optimize the performance of garbage collectors. An analysis of the memory usage was conducted for several programs in the SPECjvm98 benchmark suite. They found that Java objects are relatively small, they tend to live longer than objects in other object-oriented languages and more than 50% of the allocated space is represented by non-pointer data.

These studies all examined Java workloads, based upon specific and diverse objectives. However, all of them used Java client applications. This is in contrast with our workloads, which are based on Java server applications. In addition, we plan to focus on possible optimizations from compilers, architecture and systems. While there is some high-level overlap between our

optimization objectives and those of a few of these studies, our potential optimizations are not limited to compiler optimizations to exploit code reuse or architectural designs to optimize branching.

6. Conclusion and Future Work

This worked presented is a characterization of Java server benchmarks. The results show that the most frequently invoked methods are from the Java classes, particularly, `java.lang`, `java.util` and `java.io`. Also, within `java.lang`, the most frequently invoked methods are from the `String` class. Calling frequencies were more closely matched between jBOB and JWeb2 than between either one and pBOB. This reflects design choices in the benchmarks. Given the design choices, and the results here, it seems likely that the pattern frequency of method calls for pBOB will also differ from real Java server applications.

This work represents only the commencement of a meaningful characterization of Java server software. The next step is to augment the benchmarks studied with real applications under development. This will address the crucial question of whether or not the benchmarks are representative of Java server applications. A second step is to reexamine the behavior when executing JIT'ed code. Our current results are most accurate in identifying the proportion of various method calls made. Execution times do not consider optimizations of good JITs. A third step is to find more effective metrics for characterizing the applications. We would like, for example, to find a general way to characterize the call graph generated. Further, we would like to characterize the demands on the underlying system made by Java methods. Lastly, we would like to characterize object allocation, similar to [DIECK99].

REFERENCES

[BAYLR00] Baylor, S.J., *et. al*., "Java Server Benchmarks," *IBM Systems Journal*, Vol. 38, No. 1, 2000.

[CHOW98] Chow, K., Wright, R., and Lai, K., "Characterization of Java Workloads by Principal Components Analysis and Indirect Branches," *Workshop on Workload Characterization, Micro-31*, pp. 11-19, November, 1998.

[CONTE98] Conte, M.T., Trick, A.R., Gyllenhaal, J.C., and Hwu, W-M, "A Study of Code Reuse and Sharing Characteristics of Java Applications," *Workshop on Workload Characterization, Micro-31*, pp. 3-10, November, 1998.

[DIECK99] Dieckmann, S. and Holzle, U., "A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks," European Conference on Object-Oriented Computing (ECOOP99), 1999,

[FINK00] Arnold, M, Fink, S., Sarkar, V. and Sweeney, P., "A Comparative Study of Static and Dynamic Heuristics for Inlining", *Dynamo Workshop*, January, 2000.

[JIN99] "Jinsight", International Business Machines Corporation, http://alphaworks.ibm.com/

[JPRBE99] "JProbe ServerSide Suite," KLGroup, http://www.klgroup.com/jprobe/serverside/index.html.

[OPT99] "OptimizeIt," Intuitive Systems, Inc., http://www.optimizeit.com.

[SPEC99] "SPECweb99 Benchmark," Standard Performance Evaluation Corporation, http://www.spec.org/osg/web99/.

[VOL99] "VolanoMark 2.0," Volano LLC, http://www.volano.com/benchmarks.html.

[VTUNE99] "VTune Performance Analyzer," Intel Corp., Santa Clara, CA, http://www.intel.com/vtune/analyzer/.

[WSAS99] "Web Application Servers," International Business Machines Corporation, http://www-4.ibm.com/software/webservers/.