The Object Behavior of Java Object-Oriented Database Management Systems

Chia-Tien Dan Lo, Morris Chang, Ophir Frieder and David Grossman

Department of Computer Science
Illinois Institute of Technology
Chicago, IL, 60616-3793, USA
{danlo, chang, ophir, grossman}@charlie.iit.edu

Abstract

Due to its portability and popularity for Internet applications, Java has become one of the major programming languages. The similar syntax inherited from the C language and the pure object orientated features, compared to the non-pure object-oriented C++ language, have caused Java to be a good candidate as a tool in designing object oriented systems, especially in database servers. However, the performance of the Java Virtual Machine (JVM) would be the reason that several database designers, such as Oracle, IBM, Sybase and Informix, have not fully transferred their database management system (DBMS) into Java. One of the performance indices is the responsiveness for on-line transaction processing (OLTP) which may be dominated by the garbage collection system inside the JVM. In this paper eight Java programs, jess, javac, mtrt, compress, db, db4o, smallDB and ozone, are used to study the dynamic memory behavior. The latter four programs are Java object-oriented database management systems and their object behaviors are compared to those of the former four regular Java programs. Simulation results, such as object size distribution, average object size, object live distribution and total garbage collection cycle are reported.

Index Terms— dynamic memory management, object-oriented programming, Java Virtual Machine, garbage collection, Java DBMS

1. Introduction

Java's object-oriented paradigm (OOP) is especially good for an object-oriented database system [4, 11, 16]. A small and really portable implementation suitable for the Internet makes it an ideal companion to an object-oriented database management system (OODBMS). Its Web-enabling features have attracted several relational database server developers such as Oracle, Sybase, IBM and Informix to build their front-end user interfaces using various Java approaches. However, these client/server applications may have a crucial performance impact on the garbage collection (GC) system. Studies have shown that allocation and deallocation rate in a web server during peak time can be as high as one million calls per second [12, 24]. Thus, the popularity of Java has created the need for an analysis on object behaviors for the aforementioned applications.

An in-depth analysis of object behaviors for Java object-oriented database management system is presented in this paper. Eight benchmark programs are selected in this study. These benchmarks stand for real-world applications with a variety of characteristics. For comparison purposes, four non-database management system (non-DBMS) benchmarks are chosen as a counterpart. These benchmarks are *jess*, *mtrt*, *javac* and *compress* that are part of the SPECjvm98 benchmark [19]. *db*, *db4o* [2], *smallDB* [18] and *ozone* [17] are the Java object-oriented database management system (DBMS) benchmarks where *db* is coming from SPECjvm98 and the other three are collected from the Internet. The benchmark descriptions are detailed in a later section.

For each benchmark, we study its object behavior by measuring the average object size, total allocated objects, total garbage collection cycles, object live distribution and object size distribution. For example, to identify whether a benchmark is GC intensive, we analyze the total garbage collection cycles. Since every system has different GC implementation, we also compare our results to other reports. The objective of this study is to provide the Java DBMS community with detailed data that allows researchers to predict the impact of the Java GC system when applied to a Java object-oriented database management system.

2. Related work

There are several studies on dynamic memory management and garbage collection. One of the most comprehensive surveys on garbage collection and dynamic memory management algorithms was written by Paul R. Wilson et al. [21, 22]. However, in both papers, there are no quantitative studies of object behaviors.

In [3], Dieckmann and Hölzle performed a thorough study of SPECjvm98. In their paper, metrics such as object age, size distribution, type distribution, and object alignment overhead were reported. Although four of our benchmarks are from SPECjvm98, most of the SPECjvm98 benchmarks are non-DBMS. In this paper our objective is to study the object behavior of Java DBMS programs. Moreover, in [3],

Partial support for this work is provided by the National Science Foundation through grants CCR-0098235 and CCR-0113335

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'02)
0-7695-1506-1/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY
www.manar

the definitions of metrics are different from those used in this paper such as object live span, etc.

Calder [1] studied C and C++ programs and found that frequency of dynamic memory allocation and deallocation in C++ applications can be as much as ten times higher than similar C applications. Zorn reported the object behaviors on eight large Lisp applications by an object-level runtime system simulator [23]. Unlike our approach (GC cycles), Zorn used memory reference counts as a metric for object live span. Nettles et al. introduced Oscar [HMN 97] which is a language-independent GC testbed used to analyze object allocation behavior by recording frequent heap snapshots. Hicks et al. studied the execution time using Oscar to profile SML/NJ Java applications [7, 6]. Lo et al. studied the page replacement performance in the garbage collection heap and found a modified LUR (mLRU) policy is better for the GC heap [13, 14, 15].

3. Simulation Design

The simulation design is composed of two steps: benchmark collection and memory tracing. Finding a suitable benchmark in Java is not easy because it is hard to get a large software system developed in Java for the reason that performance issues in Java have prevented developers from using it. Moreover, compiling Java packages are not trivial. Although Java promises programmers "write once, run anywhere", some Java packages coming with a user-defined build system are hard to build. For example, the *ozone* package has adopted the IBM *jikes* that is a Java compiler that supports incremental builds and *Makefile* generation. Additionally, it also uses a Java based build tool, ant, from the Jakarta project [8], in that a configuration file is coded in XML format. These third-party tools increase the complexity when building a large Java system. For a memory tracer, it has to have the ability to log information pertaining to object behaviors. There are several methods that can be applied to the memory tracer such as source code instrumentation for a JVM and Java Virtual Machine profiler interface (JVMPI) [10] provided by Sun JDK only. The JVMPI approach has been adopted in our study.

3.1 Benchmark Programs

Eight Java programs, *jess*, *javac*, *mtrt*, jack, *db*, *db4o* [2], *smallDB* [18] and *ozone* [17] are used to generate memory pattern traces. The first five programs are from the SPECjvm98 benchmark suite [19] where the benchmark programs are designed to measure the performance of Java Virtual Machine implementations. Several criteria such as high byte-code content, flat execution profile (large loops), repeatability, heap usage and allocation rate, and I-cache or D-cache misses on the reference platform are used to test JVMs. The latter four Java object-oriented DBMS benchmarks are selected to compare to the former four non-DBMS benchmarks. These DBMS benchmarks range from a small DBMS (*smallDB*) to a large scale DBMS including a single user program (*db4o*) and a client-server system (*ozone*). The detailed descriptions of the benchmark programs are summarized in Table 1.

Table 1 Descriptions of the Benchmark Programs (BPs)

BPs	Description
jess	A Java expert shell system based on NASAs CLIPS expert shell system
db	Performs multiple database functions on memory resident database
javac	The JDK 1.0.2 Java compiler compiling 225,000 lines of code
mtrt	A dual-threaded raytracer that works on a scene depicting a dinosaur
com-press	Compress/decompress program based on modified Lem-pel-Ziv method.
ozone	*ozone* is a fully featured, object-oriented client/server data-base management system completely implemented in Java and distributed under an open source license.
db4o	*db4o* - database for objects - is a fast, small-footprint Java object-oriented database management system.
smallDB	*smallDB* is a small OODBMS in Java. Its basic design is to manipulate persistent objects by using Vector, ObjectInput-Stream and ObjectOutputStream classes.

3.2 Memory Tracer

The experiments are conducted on a Redhat Linux 6.2 computer by running a JVM from Sun's Java Development Kit version 1.2.2 [9] with a tailored profiler using JVM Profiler Interface (JVMPI) [10]. The JVMPI provides a mechanism to monitor a program behavior executed by a JVM without any change to the source code of the JVM. Interesting events such as object allocation, deallocation, object size and object type are recorded in a trace file while a benchmark program is executed. The trace file is analyzed by a parser that produces statistical information. Because any JVM loads Java programs' class files (bytecode) before execution, source code for the benchmarks is not necessary. In fact, not all the source code for these benchmarks has been obtained. Since this study only focuses on the object behavior of the dynamic memory, the availability for these class files is sufficient to gather the statistical information.

4. Simulation Results

4.1 Average Object Size, Total Allocated Objects and Total Garbage Collection Cycles

Table 2 shows the average object size, total allocated objects and total garbage collection cycle for the benchmark programs. The *compress* benchmark has the largest average object size (9,320.095) among all the benchmark programs. For DBMS benchmarks, the *ozone* has the largest average object size (36.2648) among the DBMS benchmarks

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'02)
0-7695-1506-1/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY
www.manar

possibly because it is of the client/server nature. When the server receives a request from its client, a thread object needs to be created to fulfill the request. These large thread objects have contributed to increase its average object size. On the other hand, the *db* (19.62974), *db4o* (24.1507) and *smallDB* (28.9328) show insignificant difference in their average object size compared to the non-DBMS benchmarks.

Comparing total allocated objects, the *db4o* (964,701) and *smallDB* (480,147) benchmarks have much fewer objects allocated than the other benchmarks. Also, the *db* benchmark shows about 50% less allocated objects than the non-DBMS benchmarks. This shows the non-client/server DBMS benchmarks tend to have less allocated objects. However, the client/server DBMS benchmark, *ozone*, has allocated a number of objects close to that of the *jess* benchmark. The large amount of large objects required in the client/server application results confirms the results shown in [12] that object allocation rate can be as high as one million per second. Note that our results are slightly different from those reported by [3]. Different versions of JVM are the major reason that we simulate on JDK 1.2.2 while they use JDK 1.1.5.

As to the total garbage collection cycles, the benchmarks with more objects allocated tend to trigger more garbage collection cycles. For example, the *jess* (548 GC cycles) has allocated 7,939,929 objects. However, the *db* and *smallDB* have much fewer garbage collection cycles. Furthermore, the *ozone* has allocated 7,783,601 objects but only 253 GC cycles that is about 50% less than the *jess*. We believe that the client/server DBMS creates objects that are collected by the garbage collector more efficiently.

Table 2 Average object size, total allocated objects and total garbage collection cycles

	Benchmark	Average Object Size (Bytes)	Total Allocated Objects	Total GC Cycles
Non-DBMS	*jess*	27.8606	7,939,929	548
	javac	24.7163	5,943,672	80
	mtrt	13.2805	6,644,266	54
	compress	9,320.095	11,851	28
DBMS	*db*	19.62974	3,215,855	29
	db4o	24.1507	964,701	71
	smallDB	28.9328	480,147	25
	ozone	36.2648	7,783,601	253

Because most dead objects are collected, the heap, therefore, has room to accommodate new allocation objects without starting another GC cycle. On the other hand, if most objects remain in the heap after a garbage collection cycle that is triggered by an allocation failure, the benchmark program tends to have more garbage collection cycles. This leads to the analysis of the object live span which is discussed in a later section.

4.2 Object Size Distribution

The size distribution for the benchmarks is illustrated in Figure 1. All numbers in this paper are based on 4-byte aligned object sizes. It is worth noting that there is an 8-byte alignment for the object size in JDK 1.2.2. A large portion of objects are quite small. Thus, the X-axis has been set to a logarithmic scale. A 99.5 percentage of accumulated object size is shown in Table 3. For example, there are 99.5% of objects with size less or equal to 52 in the *db4o* benchmark. The results indicate that there is a significant difference between the non-DBMS benchmarks (36-276,004) and the DBMS benchmarks (44-148). Therefore, the non-DBMS benchmark may have a lager set of sizes (e.g., *compress* (276,004 in Table 3)). This behavior is also shown in Figure 1.

4.3 Object Live Distribution

Figure 2 illustrates the object live distribution. The results confirm the weak generational hypothesis that most objects die young [5]. Most object live (age) studies evaluate object live span in terms of the fraction of bytes still live versus bytes allocated [3].

In this paper, the object live span is measured in terms of garbage collection cycles. An object has live span 8 if it dies after 8 garbage cycles since created. Table 3 lists the statistics of one-GC-cycle live objects. The non-DBMS benchmarks show that 73.32% (*javac*) - 99.20% (*jess*) of objects are collected within one garbage cycle. For the DBMS benchmarks, the data show the percentages of objects collected within one garbage collection cycle range from 78.72% (*ozone*) to 99.98% (*smallDB*).

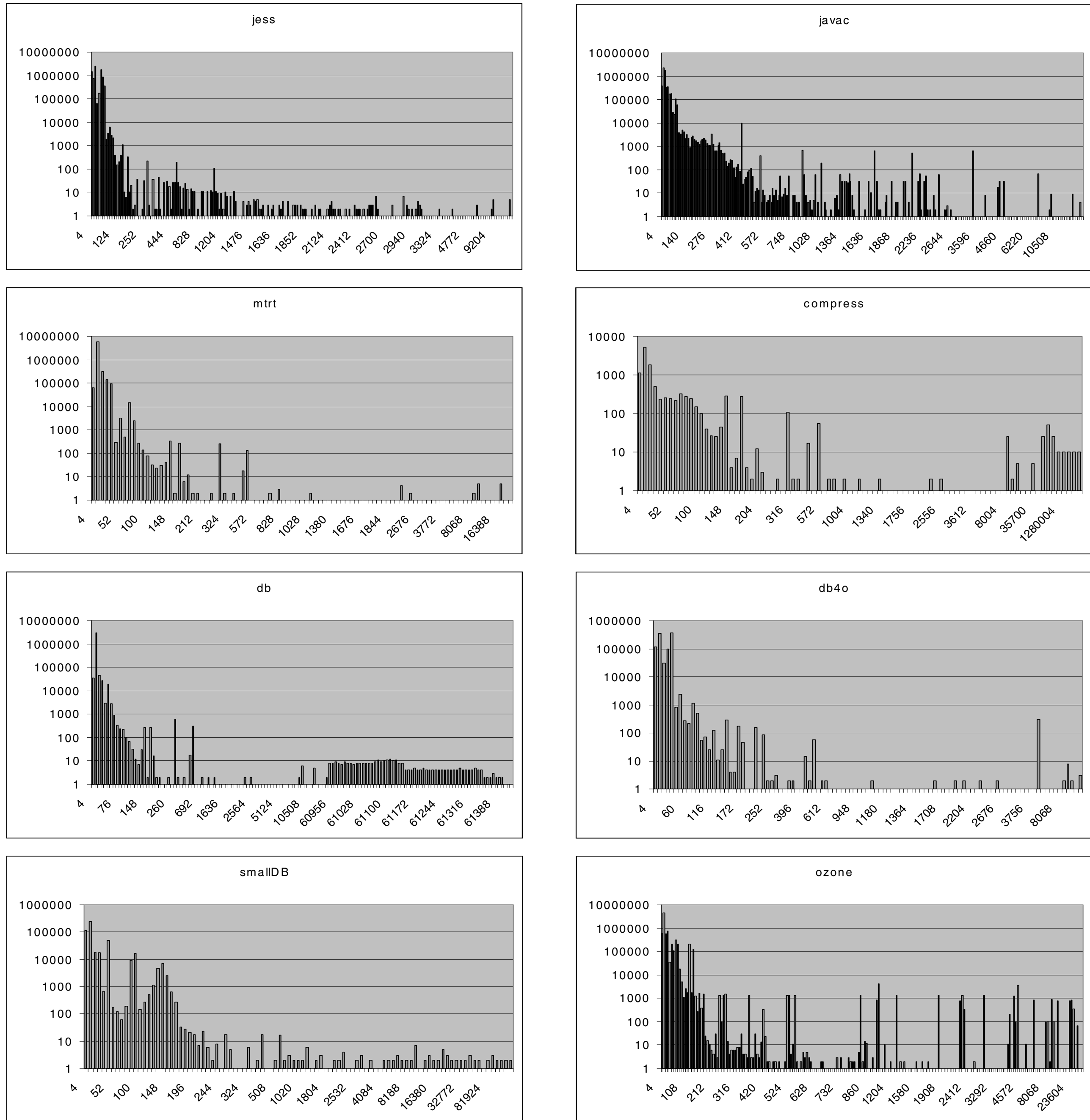
Moreover, 99.04% of objects have live span less than 55 in the *javac* benchmark; 99.02% of objects have live span less than 177 in the *ozone* benchmark. The results indicate that generational garbage collection scheme may fail in these benchmark programs because the objects may not die young. Furthermore, the *ozone* seems to have roughly 10000 objects for each live span ranging from 2 to 134.

In general, the results show that the four Java DBMS's (*db*, *ozone*, *db4o* and *smallDB*) share some similarity pertaining to the object behavior. For example, the average object size ranges from 19.63 (*db*) to 36.24 (*ozone*).

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'02)
0-7695-1506-1/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY
www.manar

Figure 1 Object Size Distribution (x-axis: object size in bytes; y-axis: number of objects)



On the other hand, it ranges from 13.28 (*mtrt*) to 9,320.095 (*compress*) for the other non-Java-DBMS benchmarks. Moreover, for the object live span, the Java DBMS's tend to create short lived objects; e.g., 99.98% objects last for one GC cycle in *smallDB* and 93.77% objects last for one GC cycle in *db4o*. However, objects may reside in the memory long for the non-Java-DBMS benchmarks; e.g., in *compress*, only 49.76% objects are collected in one GC cycle. Furthermore, we would not be surprised that most of the objects in Java DBMS's have their sizes drawn from a very small size set. For example, 94.11% of objects are drawn from the size set {4, 12, 20, 28, 44} in *ozone*. We believe that this has something to do with the way that most of the Java DBMS's store a "row" data for a database table. The typical way is mapping a "row" into an object. Therefore, most objects are allocated from this small size set.

IEEE COMPUTER SOCIETY
www.manar

.

Table 3 A 99.5 Percentage of Accumulated Object Size (bytes) and Statistics of One-GC-Cycle Live Objects

Benchmark	Non-DBMS				DBMS			
	jess	*javac*	*mtrt*	*compress*	*db*	*db4o*	*smallDB*	*ozone*
99.5% Object Size	60	236	36	276,004	44	52	148	132
One-GC-Cycle Percentage	99.1987	73.3196	95.2377	94.3505	90.7873	93.7745	99.9754	78.7238

5. Conclusions and Future Work

In this paper, we analyze the object behavior of eight Java programs including four real-world Java object-oriented database management systems and a counterpart of four real-world Java programs. The results show that the four Java DBMS's (*db*, *ozone*, *db4o* and *smallDB*) share some similarity pertaining to the object size, the object live span and object size set. The object size is similar in DBMS benchmarks, i.e., it ranges from 24.1 bytes (*db4o*) to 49.1 bytes (*ozone*). On the other hand, the object size may vary drastically. For example, it ranges from 13.2 bytes (*mtrt*) to 9442.3 bytes (*compress*) for the other non-Java-DBMS benchmarks.

Moreover, for the object live span, the Java DBMS's tend to create short lived objects; e.g., 88.76% objects last for one GC cycle in *smallDB* and 93.21% objects last for one GC cycle in *db4o*. However, objects may reside in the memory long for the non-Java-DBMS benchmarks; e.g., in *compress*, only 49.76% objects are collected in one GC cycle. Most of the objects in Java DBMS's have their sizes drawn from a very small size set. For example, 94.11% are drawn from the size set {4, 12, 20, 28, 44} in *ozone*. We believe this has something to do with the way that most of the Java DBMS's store a "row" data for a database table. The typical way is mapping a "row" into an object. Therefore, most objects are allocated from this small size set.

When migrating a database management system from a relational model to an object-oriented model, the first issue is how to map the relational model to the object-oriented model nicely for the reason that these two models have their intrinsic conflict. Performance issues such as indexing subsystem, concurrency control, recovery mechanisms and persistent object alignment remain to be studied and thus become the future work.

6. References

[1] Calder, B., Grunwald, D. and Zorn, B., 1994. Quantifying behavioral differences between C and C++ programs, Technical Report CU-CS-698-94, Computer Science Department, University of Colorado.

[2] *db4o* Java database management system, 2001. http://www.db4o.com/

[3] Dieckmann, S. and Hölzle, U., 1999. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99), Lecture Notes on Computer Science, Springer Verlag, Lisbon, Portugal.

[4] Ege, Raimund K., 1999. Object-Oriented Database Access via Reflection. In: Proceedings of the Twenty-Third Annual International Computer Software and Applications Conference., pp. 36-41.

[5] B, Hayes. Using key object opportunism to collet old objects. In Proceedings of OOPSAL'91 Conference on Object-Oriented Systems, Languages and Applications, ACM SIGPLAN Notices 26(11), Phoenix, Arizona, October 1991. ACM Press, Pages 33-46.

[6] M. Hicks, L. Hornof, J. Moore, and S. Nettles. A study of Large Object Spaces. In Proceedings of the First International Symposium on Memory Management, Vancouver, October 1998, ACM Press, Page 138-145

[7] M. Hicks, J. Moore, and S. Nettles. The measured cost of copying garbage collection mechanisms. In Proceedings of International Conference on Functional Programming, Amsterdam, June 1997

[8] Ant, the Jakarta project, http://jakarta.apache.org/ant/index.html

[9] JDK 1.2.2, 2001. JDK 1.2.2, released by Sun Microsystems, http://www.javasoft.com.

[10] Sun JVMPI Documentation, 2001. http://www.cs.southern.edu/~javadocs/guide/jvmpi/jvmpi.html

[11] King, Nelson, 1998. Java in the database Server. DBMS, June 1998, http://www.dbmsmag.com/9806d13.html

[12] Larson, P.A. and Krishnan, M., 1998. Memory Allocation for Long-Running Server Applications. In: Proc. 1998 Int'l Symposium on Memory Management, pp. 176-185.

[13] C. D. Lo, W. Srisa-an and J. M. Chang, "Performance Analysis on the Generalized Buddy System," In IEE Proceedings, Computers and Digital Techniques, Vol. 148, No. 4/5, July/September 2001, pp. 167-175

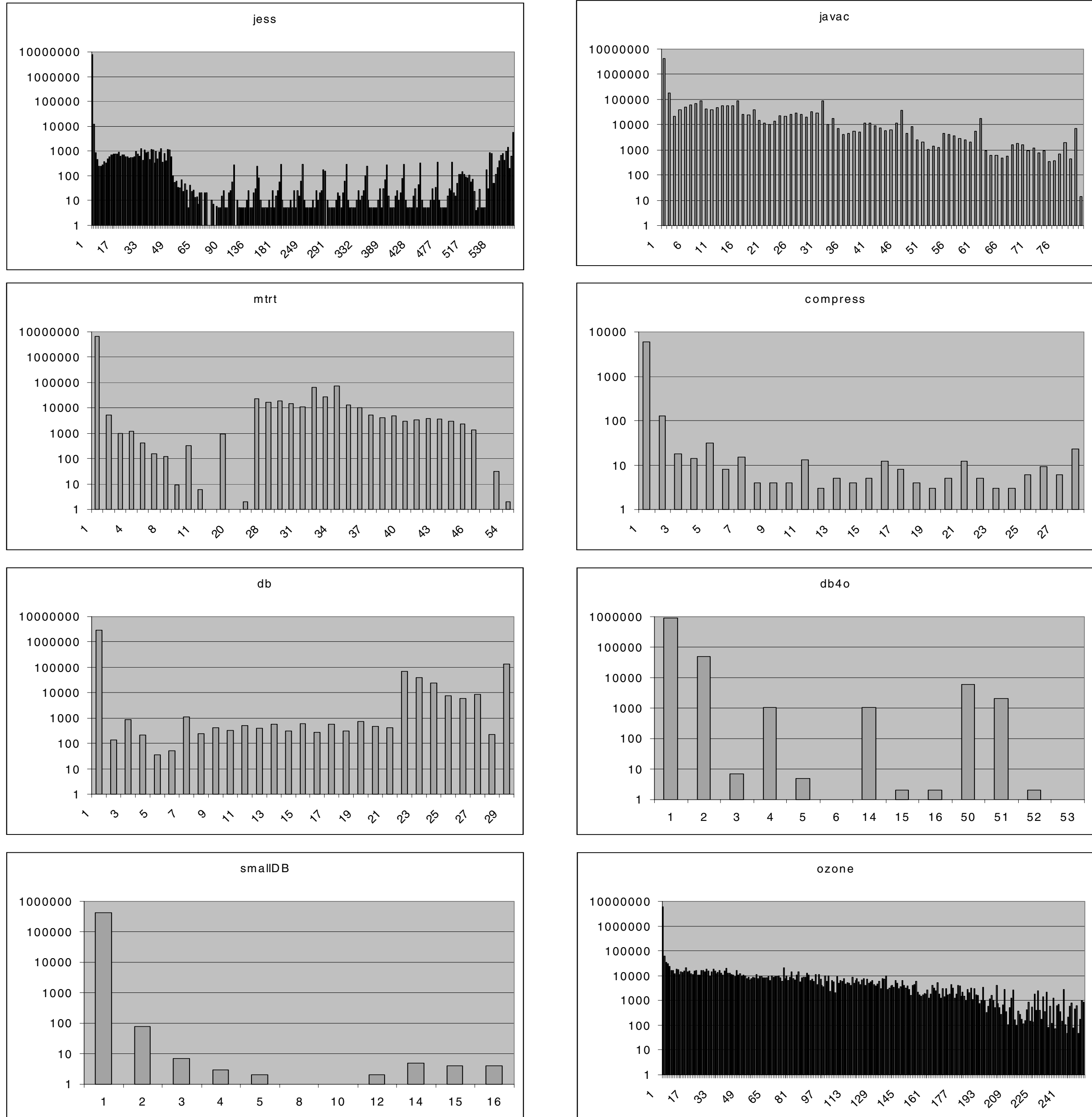
[14] C.D. Lo, W. Srisa-an, J. M. Chang, "Page Replacement Performance in Garbage Collection Systems," to appear in the Proceedings of 13th International Conference on Parallel and Distributed Computing Systems, Las Vegas, Nevada, August 8-10, 2000. pp.374-379.

[15] C. D. Lo, W. Srisa-an and J. M. Chang, "A Study of Page Replacement Performance in Garbage Collection Heap," The Journal of Systems and Software, vol. 58, 2001, pp. 235-245

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'02)
0-7695-1506-1/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY
www.manar

Figure 2 Object Live Distribution (x-axis: garbage collection cycle count; y-axis: number of objects)



[[16] North, Ken, 1999. Java in the Database. Javapro, March 1999, http://www.devx.com/upload/free/features/javapro/1999/03mar99/kn0399/kn0399.asp

[17] *ozone*, An open source Java ODBMS, http://www.ozone-db.org/

[18] A free OODBMS in Java, 2001. http://www.lifl.fr/~carono/smallDB/

[19] SPECjvm98, 1998. Standard Performance Evaluation Corporation. SPECjvm98 Documentation, Release 1.0. August 1998. Online version at http://www.spec.org/osg/jvm98/jvm98/doc/index.html.

[21] Wilson, Paul R., 1992. Uniprocessor Garbage Collection Techniques. In Proc. of International Workshop on Memory Management, 1992

[22] Wilson, P., Johnstone, M., Neely M. and Boles, D., 1995. Dynamic Storage Allocation: A Survey and Critical Review. In: Proc. 1995 Int'l workshop on Memory Management, Scotland, UK, Sept. 27-29.

[23] Zorn, B., 1989. Comparative Performance Evaluation of Garbage Collection Algorithms. Ph.D. thesis, University of California at Berkeley, March 1989.

[24] B. Willard and O. Frieder, "Autonomous Garbage Collection: Resolving Memory Leaks in Long-Running Server Applications," Computer Communications, 23 (2000) 887-900.

IEEE COMPUTER SOCIETY
www.manar