Sindh Univ. Res. Jour. (Sci. Ser.) Vol.44 (4) 667- 672 (2012)

SINDH UNIVERSITY RESEARCH JOURNAL (SCIENCE SERIES)

A Comparative Study of Garbage Collection Techniques in Java Virtual Machines

S. IQBAL, M.A. KHAN, I.A. MEMON*

Department of Computer Science, Bahauddin Zakariya University, Pakistan.

Received 07th February 2012 and Revised 12th July 2012

Abstract: Garbage collection mechanisms implemented in Java Virtual Machines (JVMs) are used to reclaim memory allocated to the objects that become inaccessible during program execution. An efficient garbage collection mechanism certainly facilitates in smooth running of an application.

This paper performs a comparative analysis of garbage collection techniques implemented in Sun HotSpot, Oracle JRockit and IBM J9 virtual machines. We perform experiments with several benchmarks containing dynamic creation for objects of TreeMap, ArrayList and String array data types. Due to the creation of a large number of big sized memory objects and loss of references, the garbage is created that is collected by the garbage collectors implemented in the virtual machines.

The experiments are carried out on different hardware architectures including Intel Core 2 Quad and Intel Xeon based systems. The performance of each garbage collector is computed in terms of the rate of garbage collection that mainly depends upon the strategy implemented for garbage collection. Our results show that the Sun HotSpot garbage collector performs 4.66 times better than the Oracle JRockit and 144.06 times better than the IBM J9 garbage collectors.

Keywords: Garbage Collection, Java Virtual Machines, JRockit, IBM J9, Sun HotSpot, Bytecode.

1. INTRODUCTION

Java is a widely used language that is implemented on a large variety of platforms ranging from distributed applications to small embedded system applications. The programs written in Java are compiled into an intermediate form called *bytecode* which is executed by a Java Virtual Machine (JVM) on a particular platform. The JVM (or simply a virtual machine) provides the environment required for execution of the Java bytecode (Oracle-Corp., 2011-a). For managing the runtime memory allocation and de-allocation, JVM uses garbage collection mechanism. JVM garbage collector searches for objects that exist independently and cannot be used by the Java programs being executed by the JVM. Such objects are treated as garbage and their memory needs to be reclaimed by the JVM. As the objects are created dynamically (at runtime), the garbage collection mechanism ensures the de-allocation of lost objects by freeing the memory for use of the newly created objects. The activities required for garbage collection at runtime impact the performance of the application (Sun-Inc., 2003; Oracle-Corp., 2011-b). An efficient garbage collection certainly makes the programs execute efficiently on a virtual machine. The performance of each garbage collector differs as the existing implementations of JVMs use different implementation strategies for garbage collection (Oracle-Corp., 2011-b; Jones and Lins, 1996; Holzle, 1993; Seligmann and Grarup, 1995; Goetz, 2003; Borman, 2002).

This paper contributes towards a performance analysis of the garbage collection approaches implemented in virtual machines. We use three benchmarks TreeMap (Oracle-Corp., 2010-a), ArrayList (Oracle-Corp., 2010-b) and String array (Oracle-Corp., 2010-c) that contain the creation of objects of the corresponding type. The objects created in these benchmarks have diverse lifetimes. Upon loss of their reference, the objects become garbage and the space allocated to them must be reclaimed in order for virtual machines to execute the programs smoothly. We analyze the performance of a garbage collector by finding the rate of garbage collection during program execution.

The remainder of the paper is organized as follows. In Section 2, we discuss the garbage collection strategies implemented by the Sun HotSpot (Paleczny et *al.*, 2001; Sun-Inc., 2001; Sun-Inc., 2010-a; Sun-Inc., 2010-b; Sun-Inc., 2006), Oracle JRockit (Oracle-Corp., 2011-c; Oracle-Corp., 2011-d) and IBM J9 (Bailey et *al.*, 2011) virtual machines. Section 3 discusses the benchmarks used for evaluating the garbage collectors. Section 4 provides the configurations used for performing experiments. The performance results are presented in Section 5 before the conclusion and future work in Section 6.

2. Garbage Collection in JVMs

The Java language provides an automatic memory management system called garbage collection

++Corresponding author: Corresponding author e-mail: mik@bzu.edu.pk
** Department of Physics, Univeristy of Sindh , Jamshoro. Pakistan

through which it lets live objects exist in memory and storage of dead objects be reclaimed. The three virtual machines Sun HotSpot (Paleczny *et al.*, 2001), Oracle JRockit (Oracle-Corp., 2011-d) and IBM J9 (Bailey *et al.*, 2011) manage the memory as a heap (a tree based data structure) and use generations for processing the objects. In this regard, *Young* generation keeps short lived objects whereas long sized objects or long lived objects are kept in *Tenured* generation.

The Sun HotSpot VM comes in two variants: Server and Client with each being selected automatically depending on the available resources of the underlying architecture such as the number of processors and memory (Sun-Inc., 2001). The JVM categorizes portions of heap as *Permanent*, *Young* and *Tenured*. Classes, methods and other code components are stored in *Permanent* area. *Young* generation collector works by using several parallel threads and copies objects in a depth-first order. When object memory area overflows, the live objects are copied to another area and the memory of the dead objects is reclaimed. The objects that survive garbage collection multiple times are then moved to the *Tenured* generation whose garbage collection incorporates the mark-and-sweep strategy. Using this approach, all accessible objects are marked and the heap memory is traversed to reclaim all the unmarked objects. The *Permanent* generation, in contrast, operates on objects that are needed by the JVM itself.

Oracle JRockit also uses heap for memory allocation. The size of heap may be changed dynamically. The JRockit VM uses a *Nursery* and a *Tenured* space for objects. Just like the Sun HotSpot JVM, it also incorporates the mark-and-sweep strategy. All the objects in use are marked, whereas the remaining objects are declared as garbage. The garbage collector operates in a parallel mode where several operations are performed simultaneously by multiple threads and consequently, the garbage collector works very efficiently. The objects with a longer life in the *Nursery* space are promoted to *Tenured* generation for possible garbage collection in subsequent operations. Furthermore, JRockit behaves differently for objects depending upon their size. The objects with large size may be directly allocated to *Tenured* area thereby avoiding frequent garbage collection for such objects.

Similar to the Sun HotSpot JVM, the IBM J9 virtual machine also uses generational heap comprising two heap areas: *Nursery* and *Tenured*. The *Nursery* area is further partitioned into *allocate* and *survivor* space Initially, the objects are allocated memory in *allocate* area, whereas the *survivor* space remains free. When memory area designated as *allocate* gets filled, the garbage collection takes place and all the live objects are copied with compaction to the *survivor* space. Consequently, the *allocate* space gets freed and roles of both the areas are interchanged, i.e. *allocate* area works as *survivor* and vice-versa. Proceeding in the similar fashion, the garbage collection continues and the long lived objects that survive for a long time are moved to the *Tenured* area for subsequent garbage collection.

3. Benchmark Description

In this section, we present the benchmarks together with the data structures and the strategy used for implementation of these benchmarks.

3.1 Data Structures

In order to incorporate a large number of memory operations with different lifetime and size of objects, we have used TreeMap, ArrayList and array of String objects as the base classes for creating the objects. The TreeMap class is part of the Java Collections Framework. It implements the SortedMap interface to support storing key to value mapping in sorted form. We make use of objects of the String class for both the keys and the values and store data using the *put* method provided by the class.

The ArrayList class implements the List interface to store objects as elements in a dynamic array. We store String elements in the ArrayList objects using the *add* method. Similarly, the third benchmark creates arrays of Strings as objects. For each benchmark, the strings are created by passing literals to the constructor of String class. As String objects are constants and could be shared, random numbers are concatenated with the literal string to keep each string separately in memory.

3.2 Benchmark Design

All the benchmarks using the TreeMap, ArrayList and String array generate a large number of objects at run time which are subsequently converted into garbage at different time intervals as given below.

- *BOSL = Big Object With Small Life*
- *BOLL = Big Object With Large Life*
- *SOSL = Small Object With Small Life*
- *SOLL = Small Object With Large Life*

The template used for generating the workload takes three arguments. The first argument CType specifies the class type of the workload i.e. TreeMap, ArrayList or String. The second argument COUNT specifies the number of iterations for which the workload should execute. Four types of objects are created within each workload. The third argument X controls the number of random strings inserted in the objects as given in the template in **(Table 1)**.

Table 1: Template for generating workloads of the benchmarks used for garbage collection

```
TEMPLATE (CType, COUNT, X)
BEGIN
    FOR I = 1 to COUNT
        Create BOLL object of class CType
        Create SOLL object of class CType
        Insert X random strings into BOLL
        FOR J = 1 to 10
        Create SOSL object of class CType
                Create BOSL object of class CType obj
                Insert random string into SOSL
                Insert X random strings into BOSL
        END FOR
END FOR
END TEMPLATE
```

Using the template, three workloads (Corresponding to TreeMap, ArrayList and String array) are generated. The code for each work load is set to execute for COUNT number of iterations. Each iteration uses the same reference for creating objects, so that for the next iteration, the previous objects become inaccessible. Within each iteration, the objects of all the types (BOSL, BOLL, SOSL, SOLL) are created. Each BOLL/SOLL object is loaded with X number of String objects generated randomly. Similarly, the inner loop creates BOSL and SOSL objects, and puts X random strings into BOSL and 1 random string into SOSL. This strategy ensures that we are generating objects that have diverse lifetimes and sizes.

Experimental Setup

The configurations used for our experimentation for evaluating the garbage collectors are given in **(Table 2)**. All the workloads are run on the Sun HotSpot, Oracle JRockit and IBM J9 JVMs with their default settings.

Table 2: Description of platform configurations used for experimentation

Sr. No	Architecture	JVMS used	Operating System	RAM
1.	Intel Core 2 Quad i7 Q720 6 GB	Java HotSpot 64-bit Server VM 1.7.0_2 Oracle JRockit JDK 28.2.2 J9 JVM with IBM SDK Version 1.4.2	Windows-7 64-bit	6GB
2.	Intel Xeon x5560 2.8 GHz dual processor		Windows Server-2008 64-bit	12GB

All the benchmarks are executed for the number of iterations ranging from 10^4 to 10^5, with an increment of 10000 together with X=40 for each run. Corresponding to an iteration size (for each run of the benchmarks), we compute the garbage collected in Kilobytes (KBytes) memory together with the collection time in seconds. To infer this information related to the performance of garbage collector, we execute the benchmark with JVM standard argument *-verbose:gc* given on command line.

Performance Results

The performance results for garbage collection are presented and analyzed in this section. For execution performance, we use the metric of Garbage Collection Ratio (GCR), defined as:

$$GCR = \frac{(Garbage\ Collected\ in\ KBytes \times Number\ of\ Iterations)}{Time\ in\ seconds}$$

Due to very small values obtained by the J9 garbage collector, we have used the logarithmic scale in performance graphs.

5.1 Intel Core 2 Quad

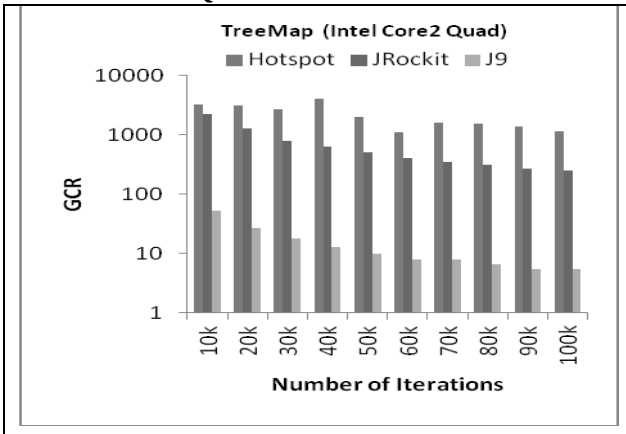

Fig. 1: GCR obtained for the TreeMap benchmark on Intel Core 2 Quad based system

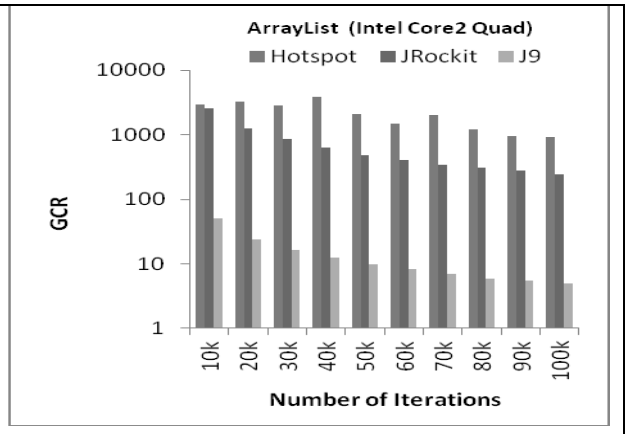

Fig. 2: GCR obtained for the ArrayList benchmark on Intel Core 2 Quad based system

For the Intel Core 2 Quad based system, the results of GCR corresponding to different number of iterations for the TreeMap, ArrayList and String array benchmarks are given in **(Fig, 1, 2,3 and 4)** respectively.

From **(Fig. 1),** it is evident that the HotSpot garbage collector for the TreeMap benchmark outperforms the garbage collectors of JRockit and J9 virtual machines.

On average, the HotSpot garbage collector attains a GCR of 2203.7 KBytes/second, whereas the JRockit and J9 garbage collectors obtain GCR values of 711.93 KBytes/second and 15.54 KBytes/second respectively. The HotSpot garbage collector therefore performs 3.09 times better than the JRockit garbage collector and 141.78 times better than the J9 garbage collector.

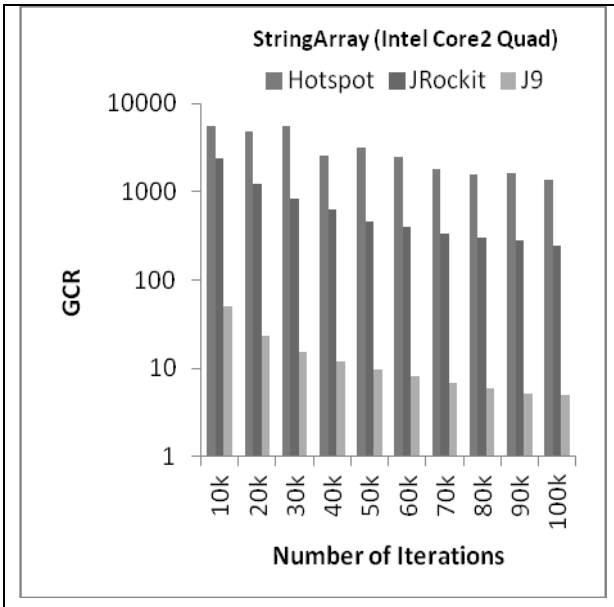

Fig. 3: GCR obtained for the String Array benchmark on Intel Core 2 Quad based system

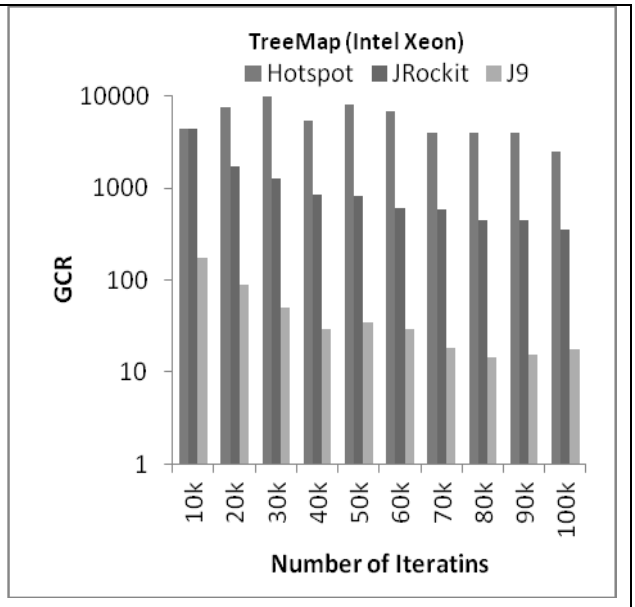

Fig. 4: GCR obtained for the TreeMap benchmark on Intel Xeon based system

Similarly, for the ArrayList benchmark **(Fig. 2),** the HotSpot, JRockit and J9 garbage collectors have average GCR values of 2178.16, 740.85 and 14.44 KBytes/second. Consequently, the Hotpsot garbage collector works 2.94 and 150.85 times better than the garbage collectors of JRockit and J9 respectively. The results are very similar for the String array benchmark **(Fig. 3)** where the HotSpot, JRockit and J9 garbage collectors have average GCR values of 3060.57, 719.70 and 14.23 KBytes/second respectively. Therefore, for this benchmark as well, the HotSpot garbage collector outperforms other garbage collectors and performs 4.25 and 215.15 times better than the JRockit and J9 garbage collectors.

5.2 Intel Xeon X5560

For the Intel Xeon based system, the results of GCR corresponding to different number of iterations for the TreeMap, ArrayList and StringArray benchmarks are given in Fig. 4, 5 6 respectively. From **(Fig. 4),** it is evident that the HotSpot garbage collector for the TreeMap benchmark outperforms the garbage collectors of JRockit and J9 virtual machines. On average, the HotSpot garbage collector attains a GCR of 5691.69 KBytes/second, whereas the JRockit and J9 garbage collectors obtain GCR values of 1151.53 KBytes/second and 46.89 KBytes/second respectively. The HotSpot garbage collector therefore performs 4.94 times better than the JRockit garbage collector and 121.38 times better than the J9 garbage collector.

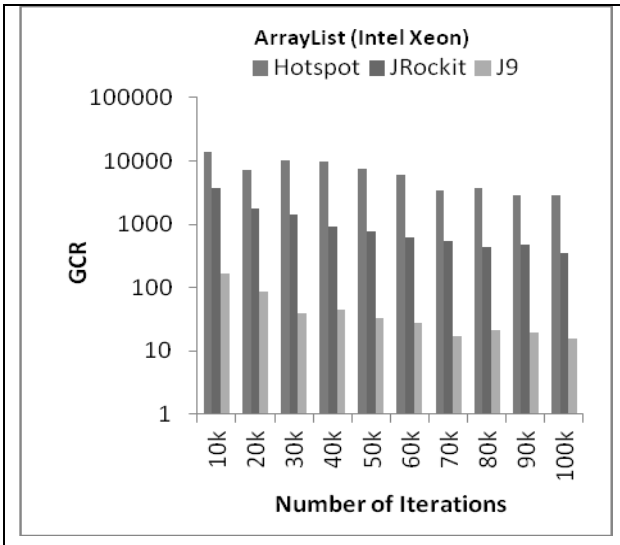

Fig. 5: GCR obtained for the ArrayList benchmark on Intel Xeon based system

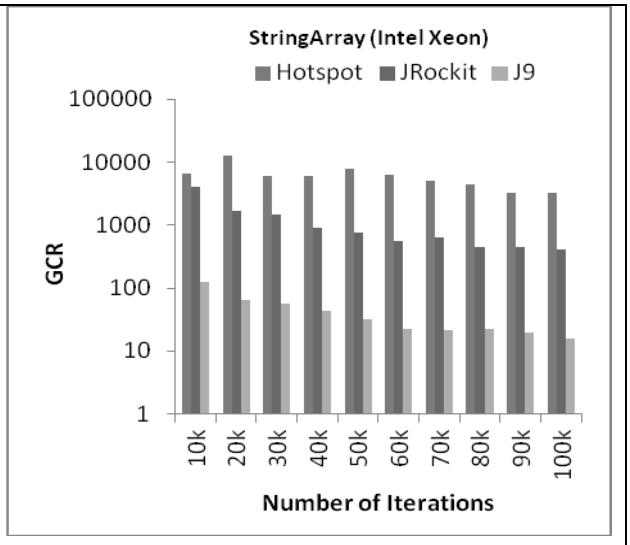

Fig. 6: GCR obtained for the String Array benchmark on Intel Xeon based system

Similarly, for the ArrayList benchmark **(Fig. 5),** the HotSpot, JRockit and J9 garbage collectors have average GCR values of 6767.35, 1113.71 and 47.29 KBytes/second. Consequently, the Hotpsot garbage collector works 6.08 and 143.11 times better than the garbage collectors of JRockit and J9 respectively. The results are very similar for the String array benchmark **(Fig. 6)** where the HotSpot, JRockit and J9 garbage collectors have average GCR values of 6121.71, 1145.24 and 42.26 KBytes/second respectively. Therefore, for this benchmark as well, the HotSpot garbage collector outperforms other garbage collectors and performs 5.35 and 144.86 times better than the JRockit and J9 garbage collectors.

5.3 Overall Summary

Considering overall performance for all the benchmarks on both the Intel Core 2 Quad and Intel Xeon based systems, the HotSpot, JRockit and J9 garbage collectors achieve GCR values of 4337.2, 930.5 and 30.1 KBytes/second respectively. Consequently. the overall performance of HotSpot garbage collector is 4.66 and 144.06 times better than the JRockit and J9 garbage collectors.

6. CONCLUSION

In this paper, we present a comparative analysis of garbage collectors of the Sun HotSpot, Oracle JRockit and IBM J9 virtual machines. We used the Intel Core 2 Quad and Intel Xeon X5560 based systems for evaluating the performance. Three different benchmarks creating garbage objects of types TreeMap, ArrayList and StringArray were developed for experimentation with garbage collectors. The benchmarks generate different objects with short or long lifetime. Upon loss of reference the objects become garbage. For evaluating the garbage collectors, we used the metric of GCR that represents the garbage collected per second per iteration.

We find that the Sun HotSpot garbage collector outperforms other garbage collectors. It achieves an average GCR of 2480.81 KBytes/second for Intel Core 2 Quad, and 6193.59 KBytes/second for Intel Xeon based system. In contrast, the Oracle JRockit and IBM J9 garbage collectors achieve average GCR values of 724.16 KBytes/second, 1136.82 KBytes/second for Intel Core 2 Duo and 14.74 KBytes/second, 45.78 KBytes/second for Intel Xeon based system respectively. Overall, the Sun HotSpot garbage collector performs 4.66 and 144.06 times better than the JRockit and J9 garbage collectors respectively.

As future work, we intend to analyze the performance of garbage collectors for distributed applications running on heterogeneous platforms.

REFERENCES:

Bailey C., C. Gracie and K. Taylor (2011) "Garbage collection in WebSphere Application Server V8, Part 1: Generational as the new default policy", IBM Developer Works, Available: http://www.ibm.com/developerworks/websphere/techjournal/1106_bailey/1106_bailey.html?ca=d rs-, USA.

Borman S. (2002) "Sensible Sanitation -- Understanding the IBM Java Collector", IBM Developer Works, Available: http://www.ibm.com/developerworks/ibm/library/ i-garbage1/, USA.

Goetz B. (2003) "Java Theory and Practice: Garbage Collection in the HotSpot JVM, Generational and Concurrent Garbage Collection", IBM Developer Works, Available: http://www.ibm.com/developerworks/java/library/j-jtp11253/, USA.

Holzle U. (1993) "A Fast Write Barrier for Generational Garbage Collectors", In Proceedings of the OOPSLA'93 Garbage Collection Workshop, USA.

Jones R. and R. Lins (1996) "Garbage Collection: Algorithms for Automated Dynamic Memory Management", Wiley and Sons, USA.

Oracle-Corp., (2011-a) "About the Java Technology", JavaTutorial,Available: http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html, USA.

Oracle-Corp., (2011-b) "Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Technology", Java Documentation, Available: http://www.oracle.com/technetwork /java/javase/gc-tuning-6-140523.html, USA.

Oracle-Corp., (2010-a) "TreeMap class", Documentation for Java(tm) 2 SE v1.4.2, Available: http://docs.oracle.com/javase/1.4.2/docs/api/java/util/TreeMap.html, USA.

Oracle-Corp., (2010-b) "ArrayList class", Documentation for Java(tm) 2 SE v1.4.2, Available: http://docs.oracle.com/javase/1.4.2/docs/api/java/util/ArrayList.html, USA.

Oracle-Corp., (2010-c) "String class", Documentation for Java(tm) 2 SE v1.4.2, Available: http://docs.oracle.com/javase/1.4.2/docs/api/java/util/String.html, USA.

Oracle-Corp., (2011-c) "Understanding Memory Management", OTN Diagnostic Guide, Available: http://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html, USA.

Oracle-Corp., (2011-d) "Oracle JRockit, Performance Tuning Guide, Release R28", Oracle JRockit Documentation, Available: http://docs.oracle.com/cd/E15289_01/doc.40/e15060/toc.htm, USA.

Paleczny M., Vick C. and Click C., (2001) "The Java HotSpot Server Compiler", In Proceedings of the JVM Research and Technology Symposium, USENIX Association, USA.

Seligmann J. and Grarup S., (1995) "Incremental Mature Garbage Collection Using the Train Algorithm", In Proceedings of ECOOP'95, LNCS, Vol. 952, pp 235-252, Springer Verlag, Germany.

Sun-Inc., (2003) "Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine", Java Documentation, Available: http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html, USA.

Sun-Inc., (2001) "The Java HotSpot Performance Engine Architecture", Sun Developer Network, Available:http://java.sun.com/products/HotSpot/whitepaper.html, USA.

Sun-Inc., (2010-a) "Ergonomics in the 5.0 Java Virtual Machine", Sun Developer Network, Available: http://java.sun.com/docs/HotSpot/gc5.0/ergo5.html, USA.

Sun-Inc., (2010-b) "Java Platform Performance Engineering", Sun Developers Network, Available: http://java.sun.com/performance/reference/whitepapers/6_performance.html, USA.

Sun-Inc., (2006) "Memory Management in the Java HotSpot™ Virtual Machine", Sun Developers Network, USA.