Performance Analysis and Problem Determination in SOA Environments

Vijay Mann[1], Venkateswara R. Madduri[1], and Srividya Shamaiah[2]

[1] IBM Research - India
{vijamann,vmadduri}@in.ibm.com
[2] IBM India Software Labs, Java Technology Center (JTC)
sshamaia@in.ibm.com

Abstract. SOA environments are typically characterized by large number of frameworks. These frameworks stack over each other in the runtime infrastructure and result in deep stack depths and large number of objects being created, most of which, are short lived. Consequently, problem determination and performance tuning of such runtime environments is known to be an extremely difficult task, which requires experience and expertise. In this paper, we share our experiences working with such production SOA runtime environments. Through our experiences we try to find the answer to the following question: can problem determination and performance analysis itself be offered as a service in SOA environments? We note that, in practice Java language and the associated J2EE stack remains one of the most popular runtime environment for implementing SOA. Since Java provides structured runtime logs, we seek to find patterns in those logs that can be used to automate performance analysis and problem determination in SOA environments. We describe three performance problem case studies, each of which present unique performance problems in open source benchmark and production SOA applications. All the case studies highlight the complexity associated with automated performance analysis. However, we make the case that at least part of the performance analysis process can be automated and offered as a service.

Keywords: problem determination, root cause analysis, performance tuning, garbage collection tuning, service performance tuning.

1 Introduction

Service oriented architecture (SOA) runtime environments are often characterized by large number of frameworks. These frameworks stack over each other in the runtime infrastructure. This results in deep stack depths (it is common to find stack depths of 100 or more in such environments) and large number of objects being created, most of which, are short lived. This, in turn, leads to peculiar performance problems in SOA runtime environments which are hard to diagnose. Consequently, problem determination and performance tuning of such runtime environments is known to be an extremely difficult task, which requires experience and expertise.

G. Pallis et al. (Eds.): ICSOC 2011, LNCS 7221, pp. 307–320, 2012.
© Springer-Verlag Berlin Heidelberg 2012

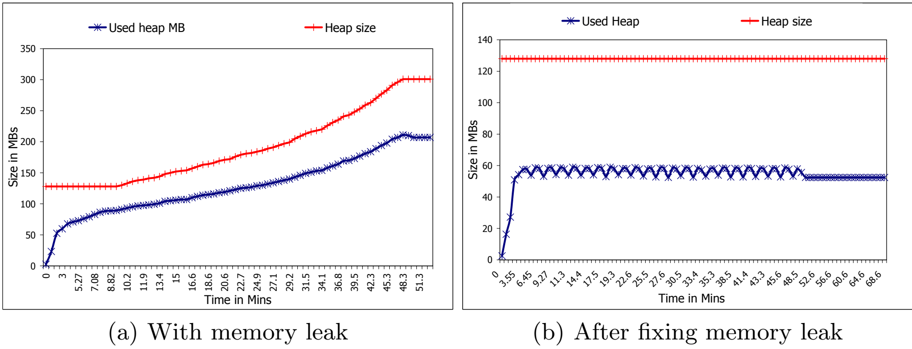
In this paper, we share our experiences working with such production and benchmark SOA runtime environments. Through our experiences we try to find the answer to the following question: can problem determination and performance analysis itself be offered as a service in SOA environments? We note that, in practice Java language and the associated J2EE stack remains one of the most popular runtime environment for implementing SOA. Since Java provides structured runtime logs, such as verbose GC logs and heap dumps, we seek to find patterns in those logs that can be used to automate performance analysis and problem determination in SOA environments. In our interactions with application development, testing and deployment teams, we found that while there was a wide array of performance tuning tools available, the knowledge and skill to use them well was not as common. Furthermore, a large time was being spent in the initial analysis of the problem, most of which can be automated. We describe three performance problem case studies, each of which present unique performance problems in an open source benchmark and a production telecom SOA application. These case studies cover a wide spectrum of performance issues: unavailability due to a memory leak in infrastructure code, high system CPU due to bad developer code, and high CPU and memory usage caused by an application design issue that got aggravated due to incorrect runtime policies and tuning. All the case studies highlight the complexity associated with automated performance analysis. However, we make the case that at least part of the performance analysis process can be automated and offered as a service.

The rest of this paper is organized as follows. Section 2 presents our first case study from an open source benchmark. Sections 3 and 4 describe the second and third case studies from a production telecom enterprise application. An overview of related research is given in Section 5. We summarize our findings and conclude in Section 6.

2 Case Study 1: Memory Leak in Infrastructure Code

In this section we present our first case study - an open source benchmark called "RUBiS" [7]. RUBiS is an open source benchmark that mimics an auction site. It comes with its own workload driver, and has a web tier that connects to a DB tier. It has been cited heavily in research papers for performance evaluation [9]. While using RUBiS as a benchmark for our other research, we noticed that RUBiS would stop responding at the end of an experiment and had to be restarted for the next experiment. We were using the Servlets version of RUBiS and MySQL JDBC driver. We analyzed the verbose GC log of the application for a constant load of 100 clients. It revealed that the used heap kept growing all the time even at constant load (refer Figure 1(a)). This seemed like a classical memory leak issue. However, as we found out, the root cause of this memory leak was not as obvious as most classical memory leaks.

We took two heap dumps of this application within minutes of each other at constant load and compared them. A quick look at the count of objects in those two heap dumps pointed us to a set of objects that have grown almost twice



(a) With memory leak (b) After fixing memory leak

Fig. 1. RUBiS heap usage at constant load

Table 1. RUBiS objects with most growth

Count	Total-size	Type	Growth/Shrink %(by count)
13,743	879,552	com/mysql/jdbc/PreparedStatement$ParseInfo	93.53
13,744	4,288,128	com/mysql/jdbc/PreparedStatement	93.52
13,727	3,294,480	com/mysql/jdbc/ResultSetImpl	93.21
25,644	4,308,192	com/mysql/jdbc/Field	90.43
42,398	2,374,288	java/util/TreeMap	89.98
74,948	2,398,336	com/mysql/jdbc/ByteArrayRow	89.81
394,707	146,608,536	byte[ ]	88.6
84,891	4,074,768	java/util/TreeMap$Entry	80.9
35,851	1,062,672	int [ ]	70.22
162,422	53,270,328	char[ ]	55.64
166,765	6,670,600	java/lang/String	54.48
60,047	2,401,880	java/util/ArrayList	32.87
63,871	2,005,904	java/lang/Object [ ]	30.35
69,266	4,433,024	java/util/HashMap	2.58
69,731	2,868,872	java/util/HashMap$Entry []	2.46
12,519	1,001,520	com/mysql/jdbc/ConnectionPropertiesImpl$Boolean ConnectionProperty	0
31,460	1,258,400	java/util/Hashtable$HashtableCacheHashEntry	-4.92
428,033	17,121,320	java/util/HashMap$Entry	-11.45
22,517	1,441,088	org/apache/tomcat/util/buf/ByteChunk	-16.93
19,856	1,111,936	org/apache/tomcat/util/buf/CharChunk	-17.02
17,646	1,552,848	org/apache/tomcat/util/buf/MessageBytes	-17.11

their original count and size. Most of these objects were of primitive types, but a few (and the ones that had grown the most - more than 90%) were related to PreparedStatement calls in JDBC. A quick breakdown of the objects that grew the most is given in Table 1.

```
Connection con = db.getConnection();
PreparedStatement stmtA;
for(int i=0;i<NUM_USERS;i++){
    stmtA= con.prepareStatement(<SOME_SQL_QUERY> );
    stmtA.setInt(1,i);
    stmtA.execute();
}
stmtA.close;
```

```
Connection con = db.getConnection();
PreparedStatement stmtA;
stmtA= con.prepareStatement(<SOME_SQL_QUERY> );
for(int i=0;i<NUM_USERS;i++){
    stmtA.setInt(1,i);
    stmtA.execute();
}
stmtA.close();
```

Fig. 2. RUBiS code leaking memory

Since we had the source code of the application with us, we searched for the usage of PreparedStatement objects and we found the pattern shown in Figure 2. Recall that a PreparedStatement object represents a precompiled SQL statement with IN parameters that can be set each time for with the specified setter methods. This way, it can be reused to execute the SQL statement multiple times efficiently. However, RUBiS code repeatedly created a lot of "temporary" PreparedStatement objects inside a loop, but closed only the instance that was created the last. JDBC specification [4] states that closing a JDBC driver should close all PreparedStatements associated with a Connection when the Connection is closed. In this case, the JDBC driver did not close the PreparedStatement objects at Connection close and maintained a reference resulting in those objects not getting garbage collected and their number kept increasing, which resulted in a memory leak.

There are two ways to fix this kind of a leak - either the statement close method is moved inside the loop or the statement creation is moved above the loop. Both will fix the memory leak, but the latter is likely to give better performance. We found that 7 out of 22 servlets in RUBiS were leaking memory at various places due to this usage pattern. The resulting heap usage graph at constant load after fixing the memory leak is given in Figure 1(b). Note that the used heap remains almost constant as expected.

This case study highlighted a memory leak scenario in infrastructure code where the leak happened due to a combination of bad usage as well as bad driver implementation. Furthermore, the root cause turned out to be something that is not usually perceived as a common cause of a leak [2]. This case study provides the following performance problem pattern:

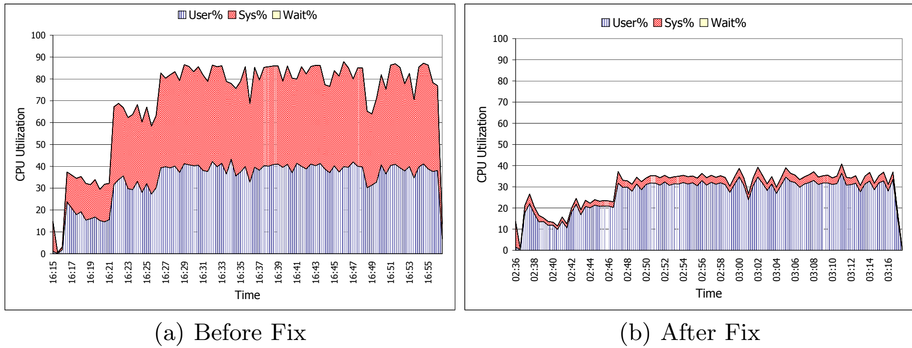
Performance Pattern 1: JDBC PreparedStatements can lead to memory leaks if each instance is not individually closed

This case study also highlighted how automated analysis of verbose GC log at constant load can detect a memory leak. A subsequent automated analysis of multiple heap dumps can be used to pinpoint the objects that are the source of this leak. The final step in the root cause detection - pin pointing the exact code block that is behind the leak, will probably still require an expert to look at the application code or thread dumps.

3 Case Study 2: High System CPU Usage

This section presents our second case study, from a SOA telecom application in production (referred to as TelecomApp in the rest of this paper). This application served as the home page for millions of mobile phone users. Users could click and get various multimedia content such as ring tones, wallpapers, live scores, etc. In production, the main components of this application consist of a cluster of web portal nodes that connect to a cluster of backend content management system nodes, which in turn get their data from a database running on a large multicore machine.

The problem that was reported was of very high CPU utilization at low to moderate number of clients. A quick look at the CPU logs revealed that the web portal node had high levels of system CPU (refer Figure 3(a).)



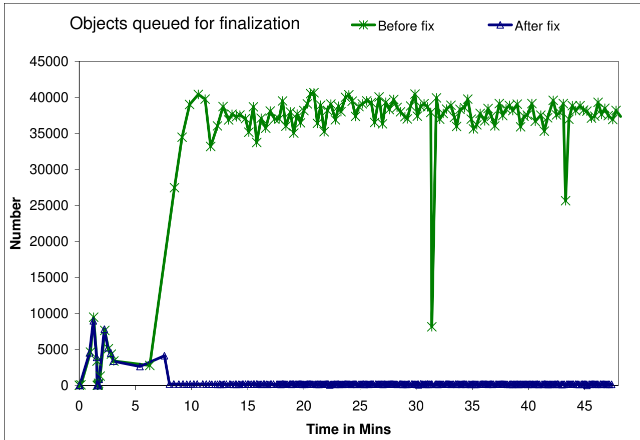
(a) Before Fix (b) After Fix

Fig. 3. Telecom App high system CPU problem

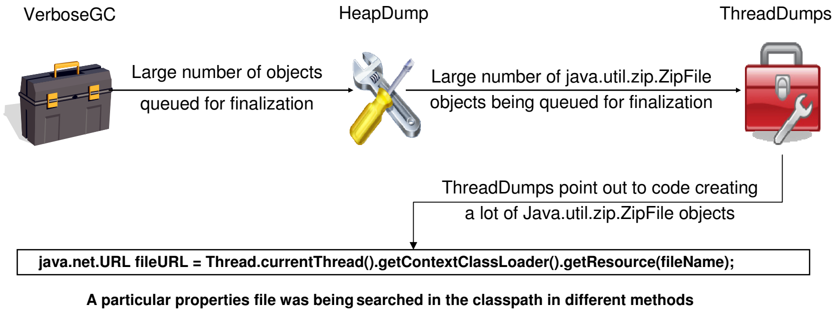
We analyzed the verbose GC logs available. GC logs looked fine except a very high number of objects being queued for finalization - close to 40000 of them (refer Figure 4(a)). Recall that all those objects that have a finalize() method and are found to be unreachable (dead) by garbage collector, are pushed into a finalization queue. At some point later, the finalizer thread will dequeue this object from the finalization queue and call its finalize() method. Too many objects being queued for finalization can cause two types of performance degradation: high CPU utilization because of the finalizer thread's work and/or high memory utilization as all the objects that are dead but reachable from an object that is yet to be finalized, can not be reclaimed.

We took a heap dump in the staging environment to find out the type of objects that were being finalized. Most of the objects that were in the finalization queue were of type "java.util.zip.ZipFile". We took threadumps that pointed us to code that was responsible for creation of these objects of type "java.util.zip.ZipFile". The code invoked the "getResource(Filename)" method on the class loader to get handle to the exact path of a particular properties file. This method, essentially, unzipped all jar files in the classpath, one by one, to search for the given filename. This resulted in creation of a lot of java.util.zip.ZipFile objects (each of which have a finalize method) and a lot of disk read activity. This unzipping of jar files and the associated disk read activity was the reason behind the observed large system CPU. This code was repeated in several methods, all of which, were on the critical path of various client requests. Steps taken to detect the root cause for this problem are shown in Figure 4(b).

We fixed this problem by replacing the code that searched for this jar file in the classpath, by the exact file location of the given file. Immediately, the system CPU went down (refer Figure 3(b)). The number of objects queued for finalization went down as well (refer Figure 4(a)). Note the initial spike in the



(a) Finalization activity in TelecomApp GC logs



(b) Steps taken to resolve TelecomApp high system CPU problem

Fig. 4. Root cause analysis of TelecomApp high system CPU problem

objects queued for finalization in both the cases - this happens as a result of all the classes being loaded from various jars at startup.

This case study highlighted a high system CPU usage scenario where the problem happened due to bad code. This case study provides the following performance problem pattern:

Performance Pattern 2: High Java object finalization activity could be a result of related high filesystem/disk activity.

This case study also highlighted how automated analysis of verbose GC log could have pointed us to abnormal finalization activity. A subsequent automated analysis of a heap dump could have been used to pinpoint the type of objects that were being finalized. The final step in the root cause detection - pin pointing the exact code block that caused the abnormal finalization activity or the high system CPU usage, would probably still have required an expert to look at the application code or thread dumps.

4 Case Study 3: High CPU and Memory Usage

In this section we present our third case study from the same TelecomApp that was described in case study 2. In this case study, we tried to solve the problem of high memory usage by the application coupled with high CPU utilization. This resulted in the application running out of heap memory after running for two or three days and halting with a heap dump getting generated. Application used the default Optthroughput GC policy in the IBM JDK which is aimed at optimal throughput.

We started with the production verbose GC log and looked for the used heap. As shown in Figure 5, the used heap curve (blue line) shows the used heap never goes below 750 M of data (even at night). The used heap becomes the lowest at night (between 1:30-6:00 am).

Logs also had a very high number of mark stack overflow errors - this happens when there are too many live objects in the heap (or more precisely very deeply nested objects) and the stack that GC uses during the mark phase overflows. Figure 5 also shows two restarts indicated by the sudden drop in heap size.

Verbose GC log also showed high pause times due to high compaction times. This pattern was very uniform though out and almost all high pause times were caused due to high compaction times (refer Figure 6). Further analysis of the logs for these high pause times, revealed allocation failures with reason code 16. Reason code 16 compaction happens when the garbage collector is not able to increase the amount of free storage by at least 10% [3]. The overhead due to garbage collection was around 13% (GC overhead=times spent in GC pauses/-total execution time * 100). Figure 6 also shows that GC keeps freeing memory - however the freed memory (green curve) keeps decreasing due to fragmentation and eventually compaction is required (blue lines). Note that heap fragmentation can also lead to "dark matter" [5] which can not be used for satisfying allocation requests. Any heap space that lies between two allocated objects and is less than 512 bytes, is not used by the JVM for allocation requests and is termed as "dark matter".

An analysis of the production heap (that got dumped on an out of memory error) showed a total of 1.2 GB of memory being used out of which close to 585 MB of the retained heap is occupied by objects belonging to two classes (refer Table 2):

1. com.TelecomApp.cache.impl.CacheEntryImpl, and
2. com.TelecomApp.mcs.devices.InternalDeviceImpl.

Note that retained size of an object is its shallow size plus the shallow sizes of the objects that are accessible, directly or indirectly, only from this object [6]. Table 3 shows the break down of the shallow heap by object types. Recollect that shallow size of an object is the amount of memory allocated to store the object itself, not taking into account the referenced objects. One can see that the top 3 contributors are all related to HashMaps. "java.util.HashMap$Entry" and "java.util.HashMap$Entry[]" contribute nearly 390 MBs (close to 33% of total heap size). This hints that both com.TelecomApp.cache.impl.CacheEntryImpl

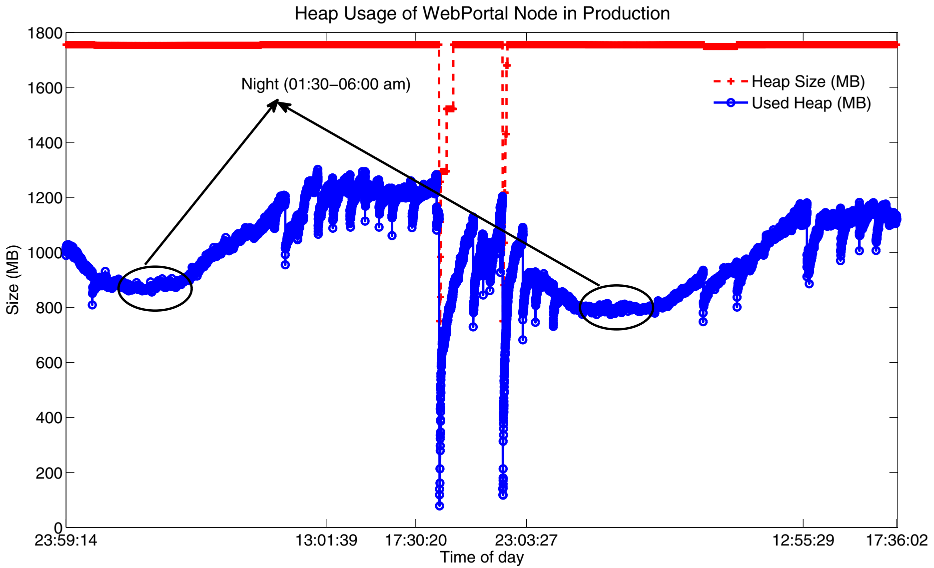


Fig. 5. Continuous high memory usage in the TelecomApp

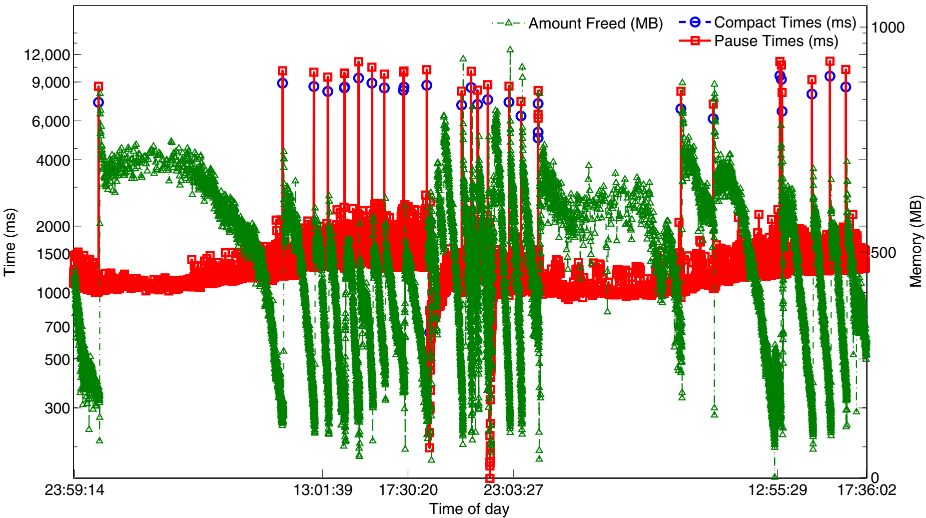


Fig. 6. TelecomApp: Pause times, compaction times and memory freed

and com.TelecomApp.mcs.devices.InternalDeviceImpl comprise of large Hashmaps. The application development team reconfirmed that the application internally uses a lot of caches to cache mobile device specific policies for faster response time. This explained, why the heap usage never goes down drastically. This seemed like an application design issue since the memory footprint of the

Table 2. TelecomApp production heap dump breakup (sorted by retained heap size)

Type	Count	Shallow Heap (MB)	Retained Heap (MB)	Retained Heap (%)
com.TelecomApp.mcs.devices.InternalDeviceImpl	8,907	0.680	418.6	33.96
com.TelecomApp.cache.impl.CacheEntryImpl	197	0.011	165.9	13.46
com.ibm.ws.cache.Cache	70	0.022	63.8	5.18
com.ibm.ws.util.ThreadPool$Worker	224	0.029	60.6	4.92
com.ibm.ws.webcontainer.webapp.WebApp	160	0.032	42.2	3.42
com.ibm.ws.webcontainer.httpsession.MemorySessionData	1,402	0.182	40.9	3.32
java.lang.Class	33,406	3.380	29.5	2.39
oracle.jdbc.driver.T4CPreparedStatement	425	0.425	27.4	2.22
com.ibm.ws.webcontainer.httpsession.MemorySessionContext	160	0.035	19.4	1.57
java.lang.String	160,233	4.890	18.3	1.48
com.TelecomApp.mcs.accessors.jdbc.JDBCDeviceRepositoryAccessor	1	0.000	15.3	1.24
com.ibm.wps.state.outputmediators.OutputMediatorFactoryProxy	96	0.003	13.6	1.1
Remainder	25,159,774	1,223	317	25.72

Table 3. TelecomApp production heap dump breakup (sorted by shallow heap size)

Count	Total-size	Type
1,871,075	305,453,504	array of char
8,558,144	273,860,608	java/util/HashMap$Entry
927,421	114,439,696	array of java/util/HashMap$Entry
1,774,002	56,768,064	java/lang/String
639,453	52,688,552	array of java/lang/Object
92,922	49,771,728	array of byte
901,808	43,286,784	java/util/HashMap
899,467	35,978,680	com/TelecomApp/mcs/themes/PropertyValue
109,740	27,300,328	array of int
26,622	16,834,352	array of com/TelecomApp/mcs/themes/StyleValue
514,376	16,460,032	java/util/Hashtable$Entry
329,675	15,824,400	com/TelecomApp/styling/properties/PropertyDetailsImpl
398,769	12,760,608	com/TelecomApp/styling/impl/engine/StylerImpl
251,689	11,886,160	array of com/TelecomApp/mcs/themes/PropertyValue
324,782	10,393,024	com/TelecomApp/mcs/css/version/DefaultCSSProperty
340,836	9,177,496	array of long
95,246	8,393,000	array of java/util/Hashtable$Entry
338,905	8,133,720	java/util/BitSet
491,191	7,859,056	java/util/HashSet
281,817	7,144,296	array of com/TelecomApp/styling/impl/engine/matchers/SimpleMatcher
284,648	6,831,552	com/ibm/ws/cache/Bucket
203,781	6,520,992	javax/servlet/jsp/tagext/TagAttributeInfo
175	5,708,528	array of org/apache/xpath/objects/Xobject
234,155	5,619,720	com/TelecomApp/styling/impl/device/DeviceStylesDelta
221,120	5,306,880	java/util/ArrayList

application seemed to be more than what the system could provide. However, the application team maintained that cached entries should automatically expire every 30 minutes or so and they ruled out a redesign since it would have required considerable effort.

Verbose GC logs also pointed that the application was using some large objects. We analyzed the heap dump for the shallow object sizes. There were only 30,000 objects greater than 1024 bytes in a total of close to 25 million objects. This constitutes almost 0.12% of the heap in terms of number of objects. However, the cumulative space occupied by these large objects was roughly 222 MB out of the total 1.2 GB (18.5%). The average object size was 51 bytes, while the median size was 32 bytes which was also the mode. This indicated that heap comprised of a very large number of small objects which were interspersed throughout the heap in between few large objects. A histogram of the shallow objects is given in Figure 7.

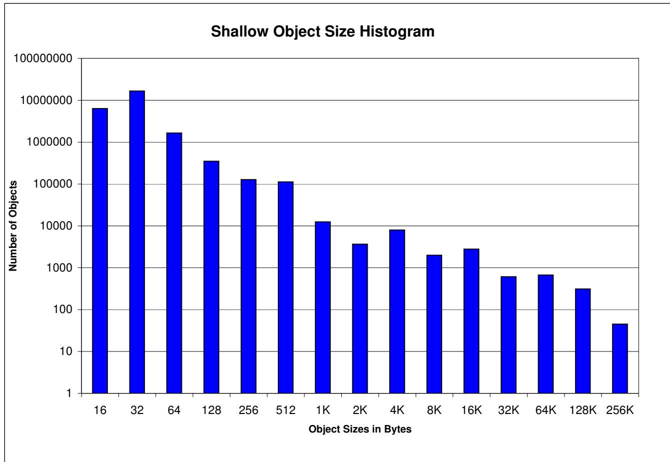


Fig. 7. Histogram of object sizes from the production heap

In order to rule out any memory leaks, a long 10 hour run at low and constant load was conducted in the staging environment. The heap usage from the verbose GC log was almost constant through out the experiment which ruled out any memory leaks. However, the staging environment could not be used to replicate the long pause times observed in the production system. This was probably due to the load offered to the system.

Our analysis so far ruled out any memory leaks and the following characteristics became apparent:

- the heap was fragmented and it resulted in high compaction times and long periods of sustained high CPU usage, as well as wasted heap space
- the application had a mix of a few large and a lot of small objects
- the application had some short-lived objects and most of the heap seemed to comprise of relatively long living cached objects.

Garbage collection techniques to resolve the above issues have been discussed in literature. Given the above application characteristics, we considered two solutions:

1. **Using a generational and concurrent collector**: This policy (known as the gencon GC policy in IBM JDK) is usually recommended for transactional applications that create a lot of short-lived objects. This policy divides the heap space into nursery (for new and short lived objects) and tenured space for old objects. Since this policy uses copying of live objects in the nursery space during a collection in the nursery (minor collection), it gets rid of fragmentation in the nursery. However,when the tenured space fills up, a global collection occurs and unreachable objects are collected, which may result in fragmentation in the tenured space. Ideally we would have liked to move all the long lived cached objects into the tenured space and leave the nursery only for short lived objects.

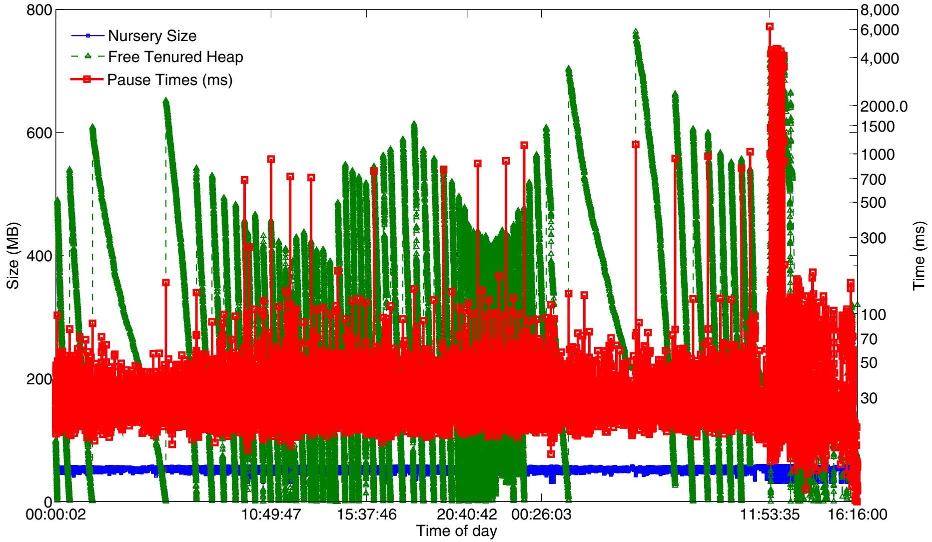


Fig. 8. TelecomApp with modified GC policy (Gencon) reduced pause times but small nursery size resulted in frequent global collections (saw-tooth pattern)

2. **Using a pool of objects** for the most frequently created objects. This resolves the problem of fragmentation by reducing creation and collection of short lived objects. However, this requires either creating a pool for frequently created objects in the application code or using the inbuilt pooling capability in the IBM JVM [1].

We tried both solutions, but realized that the first option was easier to implement on a production system. We first tried the generational and concurrent collector with default settings. Garbage collection overhead (percentage of time spent in GC pauses) went down from 13% to 1.18%. In the default gencon setting, the size of the nursery heap is set to 55-60 MB and the rest is used as the tenured space. When this setting was put into production, the high pause times went away as there was no compaction required (refer Figure 8). However, a saw-tooth pattern in the used memory and free tenured space (refer Figure 8) also indicated that the default nursery size was too small. This resulted in too many minor collections, and objects being promoted into tenured space prematurely. This also resulted in the tenured space filling up very soon and requiring a global collection.

The saw-tooth pattern revealed that around 400-550 MB of memory was being freed from the tenured space in each global collection. We used this as a yardstick to set the initial size of nursery heap to be 256 MB and the maximum nursery size to be 512 MB. When this setting was implemented in production the GC overhead further went down from 1.18% in the default gencon settings to 0.8% (refer Table 4). The problem of high CPU usage for a sustained period

Table 4. TelecomApp: performance improvement through GC tuning

GC policy	Time spent in GC pauses	Max pause time in seconds	Mean pause time in seconds
Optthroughput GC policy (original)	13.07%	11.2	1.35
Gencon GC policy (default nursery size)	1.18%	6.26	0.03
Gencon GC policy (increased nursery size)	0.8%	1.18	0.08

of time (which was being caused by frequent fragmentation and compaction in the default settings) was also resolved.

This case study highlighted a high CPU usage scenario where the problem happened due to a combination of application design and infrastructure settings (garbage collection policy settings). The application suffered high GC overheads due to heap fragmentation and repeated compaction and this resulted in high CPU utilization. This case study provides the following performance problem pattern:

Performance Pattern 3: Sustained levels of high CPU even in the absence of high load could be a result of heap fragmentation

This case study also illustrated the complexity associated with performance tuning: optimally tuning GC settings for an enterprise application that had an interesting mix of objects (mix of large and small objects, long living and short living objects) can be non-trivial and hard to automate. However, a few steps could have been automated: An automated analysis of verbose GC logs could have pointed us to high pause times which were being caused by high compaction times. Automated analysis of the heap dumps would have pointed to the existence of a large number of short-lived objects and a rule based analysis could have recommended the right GC policy for this scenario.

5 Related Research

Performance analysis and problem determination of enterprise applications is a broad area. Some performance problems are caused due to resource bottlenecks such as insufficient CPU or memory, or incorrect runtime infrastructure settings such as incorrect thread pool or connection pool sizes. These problems are somewhat independent of the nature of application and there has been a lot of work in automating problem determination and performance analysis of such problems. This typically involves statistical analysis of performance or monitoring data from middleware systems [12, 8] to detect any changes or anomalies in server performance.

On the other hand, analysis of performance problems that are closely tied to the application design, the application code, application memory usage characteristics and the interaction of the application with its runtime are much harder

to resolve and automate. The case studies presented in this paper belong to this category of performance problems. In a recent work, Chis et. al [10] present a solution that discovers a small set of high-impact memory problems, by detecting patterns within a Java heap. They demonstrate that eleven patterns cover most memory problems, and that users need inspect only a small number of pattern occurrences to reap large benefits. While this work mainly focuses on memory footprint issues and overhead of meta data in various Java collection data structures, it shares our view of using a repository of patterns to automate performance analysis.

There has also been a lot of recent work done on automating detection of memory leaks in Java programs [13, 14, 11]. While, we made use of standard heap comparison techniques to detect a memory leak in our first case study, these newer techniques can further make the process simpler and quicker and these can be part of an integrated automated peformance analysis service.

6 Conclusion

In this paper we presented our experiences working with enterprise SOA Java applications that exhibited various performance and scaling problems. We shared the insights we got while determining the root cause of some of these performance problems. We diagnosed the availability problem with RUBiS, an open source benchmark, and found the root cause to be a memory leak that occurred due to usage error as well as a non-compliant JDBC driver implementation. We presented two other case studies from a production telecom SOA application. One of them resulted in high CPU utilization caused due to code that searched a properties file in the entire classpath. This manifested itself as high number of finalization objects that get created when various jar files are unzipped. The last case study demonstrated various garbage collection issues that modern SOA applications in Java face. Through our analysis we were able to recommend the correct GC policy and tune its settings. Throughout our journey of fixing these performance problems, analysis of verbose GC logs and heap dumps proved to be valuable diagnosis tools that gave us vital clues on the probable root causes. These logs were readily available from production systems.

Our experience, also demonstrated, that the underlying causes of these problems tend to be diverse and it is very hard to provide an end-to-end root cause analysis engine for these problems. In our interactions with application development teams, we observed that they do not seem to have all the insights about the performance implications of their code and the testing and deployment teams did not have the required skill and experience to diagnose the problems from GC and heap logs. We believe that a performance analysis service for SOA environments that automates the analysis of verbose GC logs and heap dumps, and determines a high level root cause based on performance patterns such as those provided by us, can be of great value.

Acknowledgements. We would like to thank Ravi Kothari and Manish Gupta from IBM Research - India, Ashish Agrawal from IBM Global Business Services, Rajeev Palanki from IBM Java Technology Center, and Suparana Bhattacharya from IBM India Software Labs who provided us with valuable suggestions, guidance and much needed support throughout this work.

References

1. Heap fragmentation with IBM 1.3.1 and 1.4.2 JVMs, http://www-01.ibm.com/support/docview.wss?uid=swg21196072
2. Java Diagnostics Guide 1.4.2 - Common causes of perceived leaks, http://publib.boulder.ibm.com/infocenter/javasdk/v1r4m2/topic/com.ibm.java.doc.diagnostics.142/html/commoncausesofleaks.html
3. Java Diagnostics Guide 1.4.2 - verbose GC outut from a compaction, http://publib.boulder.ibm.com/infocenter/javasdk/v1r4m2/index.jsp?topic=/com.ibm.java.doc.diagnostics.142/html/id1156.html
4. JDBC 4.0 API Specification Final Release - JSR-000221, http://java.sun.com/products/jdbc/download.html
5. Mash that trash – Incremental compaction in the IBM JDK Garbage Collector, http://www.ibm.com/developerworks/ibm/library/i-incrcomp/index.html
6. Retained and Shallow Heap, http://www.yourkit.com/docs/90/help/sizes.jsp
7. RUBiS Homepage, http://rubis.ow2.org/
8. Agarwal, M.K., Sachindran, N., Gupta, M., Mann, V.: Fast Extraction of Adaptive Change Point Based Patterns for Problem Resolution in Enterprise Systems. In: State, R., van der Meer, S., O'Sullivan, D., Pfeifer, T. (eds.) DSOM 2006. LNCS, vol. 4269, pp. 161–172. Springer, Heidelberg (2006)
9. Cecchet, E., Marguerite, J., Zwaenepoel, W.: Performance and Scalability of EJB Applications. In: ACM OOPSLA (November 2002)
10. Chis, A.E., Mitchell, N., Schonberg, E., Sevitsky, G., O'Sullivan, P., Parsons, T., Murphy, J.: Patterns of Memory Inefficiency. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 383–407. Springer, Heidelberg (2011)
11. Jump, M., McKinley, K.: Cork: dynamic memory leak detection for garbage-collected languages. In: Symposium on Principles of Programming Languages, POPL (2007)
12. Mann, V., Agarwal, M.K., Gupta, M., Sachindran, N.: Problem Determination in Enterprise Middleware Systems Using Change Point Correlation of Time Series Data. In: IEEE/IFIP NOMS (2006)
13. Mitchell, N., Sevitsky, G.: Leakbot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 351–377. Springer, Heidelberg (2003)
14. Xu, G., Rountev, A.: Precise memory leak detection for java software using container profiling. In: ACM International Conference on Software Engineering, ICSE (2008)