Selection of Application Specific Garbage Collector in Client Class

Nitan S. Kotwal
Computer Science & IT
University of Jammu, India

Shubhannadan S. Janwal
Computer Science & IT
University of Jammu, India

Devanand
Computer Science & IT
University of Jammu, India

Abstract— With the advent of pure object oriented languages like Java and C# the problem of automatic storage reclamation or automatic memory management is solved by GC and there are several metrics which influence the performance of every GC. The most important metrics are Application Execution Time, Number of Pauses (minor/major), and Average time taken for each pause (minor/major). Throughput. In the current research different parameters for different types of applications have been found. These parameters can be used for setting the environment for GC before its invocation for a particular type of benchmark. The findings are based on the tests which have been performed on SPECjvm2008. Based on the results obtained, it has been proposed that selection of a particular GC and setting of the parameters for a specific application should be done before the invocation of a specific GC.

Keywords— Serial, Parallel, ParallelOld, ConcMarkSweep, GC, Benchmarks, Minor, Major.

I. INTRODUCTION

Garbage Collection is the process of automatic storage reclamation in which those objects which are no longer referenced from any live object or from program are collected. These objects are known as dead objects or garbage. The memory occupied by these objects is freed and added to the pool of free memory. This can now be allocated to the new objects. The program which accomplishes the task of garbage collection is known as GC. There are four GC’s in jdk1.7.0_04. These are Serial, Parallel, ParallelOld, and Concurrent Mark Sweep GC’s. The GC’s which is chosen by default depends on the machine architecture and operating system. For Server Class and Server Virtual Machine (VM) or Client VM the default GC is Parallel GC. For Client Class and Server VM or Client VM the default GC is Serial GC. The GC’s can also be explicitly invoked at the command line prompt by supplying commands. The various commands to explicitly call for a particular GC are: Serial GC can be invoked by –XX:+UseSerialGC. Parallel GC can be invoked by – XX:+UseParallelGC. ParallelOld GC can be invoked by –XX:+UseParallelOldGC. Concurrent Mark Sweep GC can be invoked by –XX:+UseConcMarkSweepGC. Till now only Serial and Parallel GC are automatically invoked depending on the Class of Machine and Operating system. However users can select a particular garbage collector depending on the type of application [3]. There is less or negligible effect of garbage collectors on applications that have very small amount data (kilobytes) but applications with large amount of code in execution and data (gigabytes, terabyte’s), the selection of a particular garbage collector affects the mutator. Amdahl [4] observed that all of the workload cannot be perfectly parallelized; some portion is always sequential and does not benefit from parallelism. This is also true for the Java™ platform. The various GC’s in jdk1.7.0_04 are as under:

Serial Garbage Collector: With serial collector both minor and major collections are performed serially in a stop the world fashion i.e., the application execution is stopped while the GC is running. The Serial collector is usually adequate for small applications (requiring heap up to 100 mb). [3].

Parallel Garbage Collector: With parallel collector minor collections are performed parallel with the use of multiple CPU’s in a stop the world fashion. While the major collections are performed serially in a stop the world fashion just like as performed by the serial collector. Parallel collector is suitable for those applications that have large data sets. The parallel collector is appropriate on multiprocessor systems. It is selected by default on server-class machines. It can be enabled explicitly with option -XX:+UseParallelGC[3].

ParallelOld Garbage Collector: The parallel compacting collector was introduced in J2SE 5.0 update 6. With ParallelOld collector minor as well as major collections are performed parallel with the use of multiple CPU’s in stop the world fashion. The difference between parallelOld collector and the parallel collector is that parallelOld collector uses a new algorithm for old generation garbage collection. It can be enabled explicitly with option -XX:+UseParallelOldGC[3].

Concurrent Mark-Sweep (CMS) Collector: With CMS collector minor collection are performed in the same way as performed by the parallel collector. While major collection is done concurrently with the execution of the application. The CMS collector is appropriate if application needs shorter garbage collection pauses and can afford to share processors with the garbage collector thread when the application is running. It can be enabled explicitly with option - XX:+UseConcMarkSweepGC[3].

II. REVIEW OF LITERATURE

Matthew Hertz, and Emery D. Berger[1] showed that GC’s performance improves in a large heap size. But with smaller heap size its performance degrades considerably. Sunil Soman and Chandra Krintz [2] showed that performance...
of an application is dependent on the application behaviour and available resources. They also showed that no GC performs better for all applications and heap sizes. There must be a specific GC for a particular application. Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham [6] conclude from their experiments that the execution times of various applications they tested vary significantly with the scheduling algorithm used for garbage collection. They also showed that no single configuration of the BDW collector results in the fastest execution time for all applications. The scheduling algorithm which results in fastest execution of an application also varies with the amount of memory available in the machine. Clement R. Attanasio, David F. Bacon, Anthony Cocchi, and Stephen Smith [5] observed that the mark-and-sweep collector performs better when memory is limited or when applications have a large working set as with the SPECjbb benchmark. They also showed that copying collectors perform better when objects are created at a high rate but their life span is short. They further showed that when the objects are created at a high rate and at least some of them have long lived i.e., when the working set is large then generational collectors should be used. They also proved that hybrid collector (using mark-sweep for the mature space and semi-space copying for the nursery) performs better when the memory is limited and it improves the throughput by at least 50%. Therefore hybrid collector is best for online transaction processing applications. T. Kim, N. Chang, and H. Shin [15] observed the memory system behaviour of several Java programs from the SPECjvm98 benchmark suite. They conclude that the default heap configuration used in the IBM JDK 1.1.6 results in frequent garbage collection and inefficient execution of applications. Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley [8] performed experiments, across a range of heap sizes and benchmarks, and confirm several empirical facts. First that a copying collector has better locality of reference than mark-sweep collector. The copying collector provides overall better performance when mutator time is dominant. Second the generational collectors outperform full-heap collectors by 20% on average and, in the rare cases (one out of nine benchmark) loses by 2% when the heap size is larger than 4 times the minimum heap size. Third the variable-size nursery collectors outperforms the fixed-size nursery variants by 2% among the generational collectors. Tony Printezis, and David Detlefs [9] showed that the use of mostly-concurrent algorithm for older generation decreases pauses for old-generation collection for those programs whose promotion rates are sufficiently low to allow a collector thread running on a separate processor to meet its deadlines. The young generation collection is also slowed, but this slowdown can be more than offset by the offloading of collector work to the other processor. David Detlefs [10] concludes that Real-time garbage collectors depend on upper bounds on different aspects of application behaviour, namely allocation rates and live data size. He also showed that the problem is not with the collection algorithms but with the program analysis. David Vengerov [11] showed that when the application generates relatively little garbage per unit of time, the ThruMax algorithm reduces the garbage collection (GC) overhead by more than a factor of 2, with most of the benefit coming from adaptation of the generation sizes. When the application generates a lot of garbage per unit of time, the ThruMax algorithm still reduces the GC overhead by almost a factor of 2, with most of the benefit coming from adaptation of the tenuring threshold. Finally, when the application's load was allowed to increase over time, either linearly or as a stepfunction, the ThruMax algorithm was able to quickly reduce the young generation size and avoid the situation when only the major garbage collections are occurring. Katherine Barabash, Yoav Ossia, and Erez Petrank [12] presented a modification of the concurrent collector, by improving the throughput of the application, stack, and the behavior of cache of the collector without foiling the other good qualities (such as short pauses and high scalability). They implemented their solution on the IBM production JVM and obtained a performance improvement of up to 26.7%, a reduction in the heap consumption by up to 13.4%, and no substantial change in the pause times (short). The proposed algorithm was incorporated into the IBM production JVM. Stephen M Blackburn, Perry Cheng and Kathryn S McKinley [13], experimental design shows key algorithmic features and how they match program characteristics to explain the direct and indirect costs of garbage collection as a function of heap size on the SPEC JVM benchmarks. They find that the contiguous allocation of copying collectors attains significant locality benefits over free-list allocators. The reduced collection cost of the generational algorithms together with the locality benefit of contiguous allocation motivates a copying nursery for newly allocated objects. The above mentioned advantages dominate the overheads of generational collectors compared with non-generational. Christine H. Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang [14] after implementing two parallel collectors, believed that there was great potential for improving both pause times and throughput using parallelism. Sequential garbage collection algorithms would become a bottleneck for multithreaded applications which require heap size in gigabyte. The systems which are intended to support such applications, stops all threads for garbage collection, must use parallel techniques. R. Fitzgerald and D. Tarditi [16] and C. R. Attanasio, D. F. Bacon, A. Cocchi, and S. Smith, [5] suggests there is no garbage collector that gives best performance for all the applications. i.e. different applications have different requirements of GC.

III. EXPERIMENTATION

Benchmarks

SPECjvm2008 benchmark suite is used in the current research. The brief description of the benchmarks is shown in Table 1. All the eleven benchmarks available in SPECjvm2008 are studied in real JVM. No simulators are being used in the experimentation. All the eleven benchmarks specified in the SPECjvm2008 are executed over a wide range of heap size varying from 20 mb to 400 mb with an increment of 20 mb size. Each of the benchmark is executed 10 times in a fixed heap size and the arithmetic mean is obtained. The performance of the Serial and Parallel collector is measured over different heap sizes. All the tests are performed on a fixed hardware and software.

Java used for performing the tests is jdk1.7.0_04, Ergonomics machine class is client. JVM name is JavaHoTSpot(TM) Client VM. The issues considered for optimization in the current research are...
Pauses
Pauses are defined as the temporary suspension of the mutator so that the garbage collection can take place. Pauses can be of two types
1) Minor: Initially the objects are allocated in the young generation and when this generation fills up, it causes minor collection.
2) Major: The objects which survive after the minor collection are moved to the tenured generation. When this generation fills up major collection is invoked.

Throughput
Throughput is defined as the percentage of the total time not spent in garbage collection. The throughput of an application should be more. It is calculated with the help of the following formula.

\[ \text{Throughput} = \frac{\text{mutator time}}{\text{mutator time} + \text{garbage collection time}} \times 100 \]

Pause Time in Minor Collection
It is defined as the time interval in minor collection during which a garbage collector is removing unreferenced objects from young generation. During minor pauses the mutator/application is suspended temporarily for a short period of time. It occurs when the young generation fills up.

Pause Time in Major Collection
It is defined as the time interval in major collection during which a garbage collector is collecting garbage from tenured generation. During major pauses the mutator/application is suspended for long as compared to minor pauses. It occurs when the tenured generation fills up.

Application Execution Time
This is defined as the time spent by the application/mutator during its execution including garbage collection time. The mutator execution time should be less.

### TABLE 1

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Startup</td>
<td>This benchmark starts each benchmark for one operation. Startup benchmark is single-threaded.</td>
</tr>
<tr>
<td>Compiler</td>
<td>This benchmark uses the OpenJDK (JDK 7 alpha) front end compiler to compile a set of .java files.</td>
</tr>
<tr>
<td>Compress</td>
<td>This benchmark compresses data, using a modified Lempel-Ziv method (LZW).</td>
</tr>
<tr>
<td>Crypto</td>
<td>This benchmark encrypt and decrypt data using AES (crypto.aes), RSA (crypto.rsa) and sign verification (crypto.signverify).</td>
</tr>
<tr>
<td>Derby</td>
<td>The focus of this benchmark is on BigDecimal computations, database logic, and on locks behavior.</td>
</tr>
<tr>
<td>MPEGaudio</td>
<td>This benchmark is used for mp3 decoding</td>
</tr>
<tr>
<td>Scimark</td>
<td>This benchmark is widely used by the industry as a floating point benchmark. There are two versions of this test, one with a &quot;large&quot; dataset (32Mbytes) and another with &quot;small&quot; dataset(512Kbytes).</td>
</tr>
<tr>
<td>Serial</td>
<td>This benchmark serializes and deserializes primitives and objects, using data from the JBoss benchmark.</td>
</tr>
<tr>
<td>Sunflow</td>
<td>This is a multi-threaded benchmark used in image rendering system.</td>
</tr>
<tr>
<td>XML</td>
<td>This Benchmark apply style sheets to XML documents using javax.xml.transform, and validating documents by javax.xml.validation.</td>
</tr>
</tbody>
</table>

IV. RESULTS

Startup
This benchmark starts each benchmark for one operation. A new JVM is launched and time is measured from start to end. Start up benchmark is single-threaded. This allows multi-threaded JVM optimizations at startup time. Startup launcher is required to be the same as a 'main' JVM for submissions. Both startup launcher and arguments for startup launcher can be modified. Each startup benchmark runs the suite with the single benchmark. Each non-startup benchmark is used as startup benchmark argument, except derby. Startup,scimark benchmarks use 512Kbytes datasets [17]. The application of this type should be executed with ParallelOld GC. It is initialized at 20 MB heap size. The heap size where it gives optimal performance for all the above mentioned metrics is 220 MB. The result for startup benchmark is shown in “Fig. 2”.

Compiler
This benchmark uses the OpenJDK (JDK 7 alpha) front end compiler to compile a set of .java files. The code compiled is javac itself and the sunflow sub-benchmark from SPECjvm2008. This benchmark uses its own FileManager to deal with memory rather than with disk and file system operations. [17]. This benchmark requires at least 60 MB heap size to utilize for Serial, Parallel, ConcMarkSweep GC’s and 80 MB in case of ParallelOld GC. The GC which is
best suited for its requirement is Serial GC. Serial collector gives optimal performance for all the mentioned metrics. The result for compiler benchmark is shown in “Fig. 3”.

**Compress**
This benchmark compresses data, using a modified Lempel-Ziv method (LZW). Basically finds common substrings and replaces them with a variable size code. This is deterministic, and can be done on the fly. Thus, the decompression procedure needs no input table, but tracks the way the table was built. This is a Java port of the 129.compress benchmark from CPUI95, but improves upon that benchmark in that it compresses real data from files instead of synthetically generated data as in 129.compress [17]. For Serial and Parallel GC this benchmark need 40 MB, for ConcMarkSweep GC it need 60 MB, and for ParallelOld GC it need 80 MB to initialize. This benchmark should be executed with ConcMarkSweep GC. 280 MB is the best heap size where it gives optimal performance for all the metrics mentioned above. The result for compress benchmark is shown in “Fig. 4”.

**Crypto**
This benchmark focuses on different areas of crypto and are split in three different sub-benchmarks. The different benchmarks use the implementation inside the product and will therefore focus on both the vendor implementation of the protocol as well as how it is executed [17].

1. aes encrypt and decrypt using the AES and DES protocols, using CBC/PKCS5Padding and CBC/NoPadding. Input data size is 100 bytes and 713 KB.
2. rsa encrypt and decrypt using the RSA protocol, using input data of size 100 bytes and 16 KB.
3. signverify sign and verify using MD5withRSA, SHA1withRSA, SHA1withDSA and SHA256withRSA protocols. Input data size of 1 KB, 65 KB and 1 MB.

This benchmark is initialized at 20 MB for all the collectors. Crypto benchmark should be executed with ParallelOld GC. The heap size on which this benchmark should be executed is 240 MB. The result for crypto benchmark is shown in “Fig. 5”.

**Derby**
This benchmark uses an open-source database written in pure Java. It is synthesized with business logic to stress the BigDecimal library. It is a direct replacement to the SPECjvm98 db benchmark but is more capable and represents as close to a “real” application. The focus of this benchmark is on BigDecimal computations (based on telco benchmark) and database logic, especially, on locks behavior. BigDecimal computations are trying to be outside 64-bit to examine not only 'simple' BigDecimal, where 'long' is used often for internal representation [17]. It is initialized at 220 mb for Serial collector and 240 mb for all other collectors. Derby benchmark should be executed with Parallel collector. It gives optimal performance at 400 MB for all the mentioned metrics. The result for derby benchmark is shown in “Fig. 6”.

**Mpegaudio**
This benchmark is very similar to the SPECjvm98 mpegaudio. The mp3 library has been replaced with JLayer, an LGPL mp3 library. Its floating-point heavy and a good test of mp3 decoding. Input data were taken from SPECjvm98 [17].

This benchmark is initialized at 20 MB for all the collectors. It should be executed with ParallelOld collector at 360 MB heap size. The result for mpegaudio benchmark is shown in “Fig. 7”.

**Scimark.large**
This benchmark was developed by NIST and is widely used by the industry as a floating point benchmark. Each of the subtests (fft, lu, monte_carlo, sor, sparse) were incorporated into SPECjvm2008. There are two versions of this test, one with a "large" dataset (32Mbytes) which stresses the memory subsystem and a "small" dataset which stresses the JVMs (512Kbytes) [17]. For Serial collector this benchmark is initialized at 160 MB heap size while for all other GC’s it is initialized at 180 MB heap size. This benchmark should be executed with ConcMarkSweep collector. The heap size where it gives optimal values for all the metrics mentioned above is 380 MB. The result for scimark.large benchmark is shown in “Fig. 8”.

**Scimark.small**
This benchmark is initialized at 20 MB heap size for all the collectors. Although there are large number of minor pauses at starting heap size but it decreases as the heap size increases. The ParallelOld collector is most suitable for this benchmark. It should be executed at 320 MB heap size. The result for scimark.small benchmark is shown in “Fig. 9”.

**Serial**
This benchmark serializes and deserializes primitives and objects, using data from the JBoss benchmark. The benchmark has a producer-consumer scenario where serialized objects are sent via sockets and deserialized by a consumer on the same system. The benchmark heavily stress the Object.equals() test [17]. This benchmark need at least 120 MB heap size for Serial, Parallel, and ConcMarkSweep GC’s to initialize. It needs 140 MB heap size to initialize ParallelOld collector. It should be executed with Parallel collector at 380 MB heap size for optimal values of all the metrics. The result for serial benchmark is shown in “Fig. 10”.

**Sunflow**
This benchmark tests graphics visualization using an open source, internally multi-threaded global illumination rendering system. The sunflow library is threaded internally, i.e. it's possible to run several bundles of dependent threads to render an image. The number of internal sunflow threads is required to be 4 for a compliant run. It is however possible to configure in property specjvm.benchmark.sunflow.threads.per.instance, but no more than 16, per sunflow design. Per default, the benchmark harness will use half the number of benchmark threads, i.e. will run as many sunflow benchmark instances in parallel as half the number of hardware threads. This can be configured in
specjvm.benchmark.threads.sunflow [17]. This benchmark is initialized at 20 MB heap size for all the collectors. Sunflow should be executed with ConcMarkSweep GC. This benchmark gives optimal value for all the metrics at 380 MB heap size. The result for sunflow benchmark is shown in “Fig.11”.

XML

This benchmark has two sub-benchmarks: XML.transform and XML.validation. XML.transform exercises the JRE's implementation of javax.xml.transform (and associated APIs) by applying style sheets (.xsl files) to XML documents. The style sheets and XML documents are several real life examples that vary in size (3KB to 156KB) and in the style sheet features that are used most heavily. One "operation" of XML.transform consists of processing each style sheet / document pair, accessing the XML document as a DOM source, a SAX source, and a Stream source. In order that each style sheet / document pair contribute about equally to the time taken for a single operation, some of the input pairs are processed multiple times during one operation. Result verification for XML.transform is somewhat more complex than for other of the benchmarks because different XML style sheet processors can produce results that are slightly different from each other, but all still correct. In brief, the process used is this. First, before the measurement interval begins the workload is run once and the output is collected, canonicalized (per the specification of canonical XML form) and compared with the expected canonicalized output. Output from transforms that produce HTML is converted to XML before canonicalization. Also, a checksum is generated from this output. Inside the measurement interval the output from each operation is only checked using the checksum. XML.validation exercises the JRE's implementation of javax.xml.validation (and associated APIs) by validating XML instance documents against XML schemata (.xsd files). The schemata and XML documents are several real life examples that vary in size (1KB to 607KB) and in the XML schema features that are used most heavily. One "operation" of XML.validation consists of processing each style sheet / document pair, accessing the XML document as a DOM source and a SAX source. As in XML.transform, some of the input pairs are processed multiple times during one operation so that each input pair contributes about equally to the time taken for a single operation [17]. This benchmark should be initialized at 40 MB heap size for Serial, Parallel, and ParallelOld GC's. It is initialized at 60 MB heap size for ConcMarkSweep collector. It should be executed with ConcMarkSweep GC. This collector gives optimal performance for all the metrics discussed above at 400 MB heap size. The result for xml benchmark is shown in “Fig. 12”.

After obtaining and analyzing the results it is observed that the no single Garbage collector based on different algorithm is best suited for the all the different types of application [7]. In the current paper it has been found that the different garbage collector gives different performance in different environment. Now it recommended that, particular type application should select a specific type of collector, suited for it, before its execution as shown in “Fig. 1”. By doing so, the performance of the machine can be significantly improved.

V. CONCLUSIONS

Different applications have different requirements. After the empirical investigation, different parameters for different types of applications have been found. These parameters can be used for setting the environment for GC before its invocation for a particular type of benchmark. The findings are based on the tests which have been performed on SPECjvm2008.
Keeping in view the various metrics we have to select a particular GC for an application that would result in best and optimal performance for that application.

It is proposed that the garbage collector should be activated based on application behavior. In future we also wish to find the optimal values for all the GC for DaCapo-9.12-bach benchmark suite.

![Fig.2 Result of startup.](image)

![Fig.3 Result of compiler.](image)
Fig. 4 Result of compress.

Fig.5 Result of crypto.
Fig. 6 Result of derby

Fig. 7 Result of mpegaudio.
Fig. 8 Result of scimark.large.

Number of Pauses for minor collection
Avg. Pause time for minor collection
Application execution time

Fig. 9 Result of scimark.small.

Number of Pauses for minor collection
Avg. Pause time for minor collection
Application execution time

Number of Pauses for major collection
Avg. Pause time for major collection
Throughput
Fig. 10 Result of serial.

Fig. 11 Result of sunflow
Fig.12 Result of xml.

REFERENCES


