Enhancing the Usage of the Shared Class Cache

by

Devarghya Bhattacharya

Bachelors of Information Technology, WBUT, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Kenneth B. Kent, PhD, Faculty of Computer Science
	Eric Aubanel, PhD, Faculty of Computer Science
Examining Board:	David Bremner, PhD, Computer Science, Chair
	Gerhard Dueck, PhD, Computer Science
External Examiner:	Richard Tervo, PhD, Electrical and Computer Engineering

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

October, 2016

Abstract

With the increasing popularity of the Java language and sandboxed environments, research needs to be conducted into improving the performance of these environments by decreasing the execution time as well as the memory footprint of an application. This thesis examines various critical data structures, used by IBM's *Java Virtual Machine* (JVM) during the start-up phase, for potential improvements. These data structures start small and expand as required in order to save space, however, growing them slows down the start-up of the JVM.

This thesis will describe how the data structures were optimized using the *Shared Class Cache* (SCC), in order to improve the execution time as well as the memory footprint of the application running on IBM's JVM. The impact of this approach on performance and memory has been evaluated using different benchmarks. On average, a performance increase of 6% and a memory reduction of about 1% has been achieved with this approach. The alterations made are completely automated and the user requires no prior knowledge about the Java application or the VM to improve the performance of the

deployed application. The only task the user has, is to activate the SCC.

Dedicated

To

Ma

I don't thank you enough for the sacrifices you have done for me.

Baba

For being the best father anyone can dream to have.

Jethu

Close or far away, you always took care of me.

Shubho Dada and Mou Didi

I would not be here, if not for both of you.

Acknowledgements

I would like to start by thanking my supervisors, Kenneth B. Kent and Eric Aubanel, for being there and guiding me through this degree. I have learned a lot and I am grateful for the opportunity. I would also like to thank Peter Shipton and Hang Shao from IBM Canada, for finding time amidst their busy schedule to provide help and guidance whenever I needed it. Finally, I would like to thank Stephen MacKay for all the help he provided when I was writing this thesis.

I would like to thank my lab mates Konstantin Nasartschuk, Azden Bierbraur, Federico Sogaro, Taees Eimouri and Aaron Graham, for all their support and guidance throughout my time in the CASA lab. Without their help, writing this thesis would have not been possible.

The funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program is gratefully acknowledged.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The performance of IBM's Java Virtual Machine (JVM), especially pertaining to the start-up of an application, has been an area for improvement. The following research successfully improved the performance of the JVM using the Shared Class Cache. The JVM is the middleware that runs Java applications, making it a potential bottleneck. The JVM uses several data structures that might have to be resized during the execution of the application, which is undesirable as it adds to the execution time. Therefore, new approaches need to be researched and implemented to speed up the JVM and reduce its memory footprint.

The *Shared Class Cache* (SCC), a common memory space shared among each and every JVM connected to it, was introduced in Java 5. Since then, JVMs do not have to load the same set of classes or compile every method. Instead, they can load it once and store the information in the SCC and reuse the data

in the future. Reusing the immutable information is beneficial as the JVM will not have to perform the same task again, namely dynamically growing data structures multiple times to reach the required size. This research aims to find more immutable information that can be shared among JVMs. In the following chapters, a detailed explanation about different information that was added to the SCC will be provided.

This thesis is structured as follows: in Chapter 2, the background as well as related work are provided, to explain the scope of the problem. In Chapter 3, the problem formulation and the design of this project are explained. In Chapter 4, the actual implementation is explained while its evaluation is given in Chapter 5. Finally, this thesis concludes in Chapter 6 and gives an outlook on possible future work.

Chapter 2

Background

2.1 Java

Java is a high-level, object-oriented language. It is class-based and concurrent. Java is similar to C/C++, but to make it more productive, features were introduced, such as making Java bytecode portable, which can be compiled once and executed on any platform and also, extensive libraries designed to provide a full abstraction of the underlying platform. Java was designed to deal with multiple architectures, which required the compiled code to be transferred over networks and also work on multiple client machines [20].

The main advantage of Java is that it follows the principle of "write once, run anywhere", meaning that Java code, once compiled, can be run on multiple platforms that have a Java Virtual Machine (JVM). A Java program is compiled into bytecode, which can be executed on any JVM irrespective of

the architecture, making the language portable [20]. Version 8 of Java was used for this research.

2.2 Java Virtual Machine

The *Java Virtual Machine* is the abstract computing machine that is required to execute a Java program. The JVM makes Java hardware and operating system independent and also is responsible for the small size of the compiled code. It sandboxes the Java program from the underlying operating system so that the Java program does not interfere with the operating system, hence, protecting the operating system from being harmed, and also, provides an environment that is similar for each machine [20].

The JVM has no knowledge about the Java language. It only deals with instructions and memory locations at run time. It only knows about the class file format, which is a binary file that contains Java bytecode. So, it can execute code written in any language provided it can be represented in the class file format [20].

To speed up execution time of an application, the JVM makes use of the Just-in-Time Compiler (JIT) for compiling during runtime. The JIT compiler reads the bytecode and changes it dynamically to machine language, only if it deems that there is an advantage.

Figure 2.1 shows how the JVM works. The source code (.java) is compiled by the `javac` compiler into bytecode (.class). This bytecode can then be

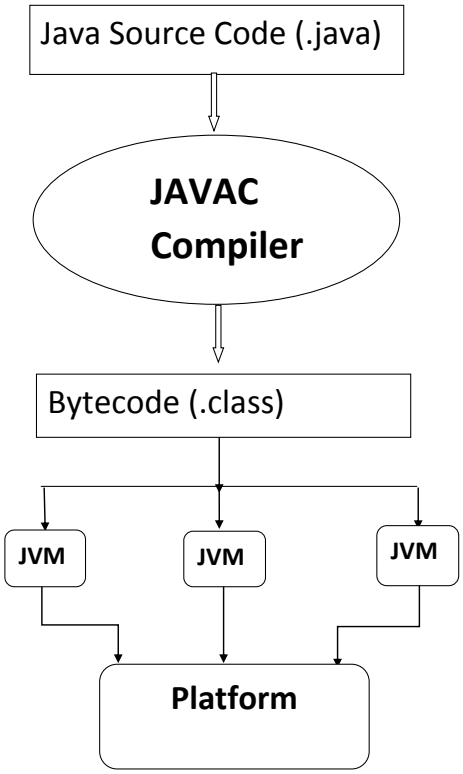

Figure 2.1: Flow of executing Java Source Code [3].

executed on any JVM that may be running on any platform. The JVM version that was used for this research is IBM's J9 VM.

2.3 ClassLoader

Class Loaders are objects of the class *ClassLoader*, used by the JVM, to dynamically find classes in memory at run time. It is provided with the binary names of the classes; once provided, it checks if the classname is valid

or not and finally, checks if the class has already been loaded. The Class Loaders try to search for a class in the cache first (to see if the class was loaded earlier) and if it fails to find the class, it searches in the file-system. Finally, it fetches the class for the JVM. The JVM has its own set of Class Loaders but they can only load classes from the local file-system and hence, custom Class Loaders are created to load classes from networks and other file-systems [17].

Every class in memory is loaded by a Class Loader and a Class Loader can load a particular class only once. The classes are generally loaded on demand. In Java, libraries are stored as Java Archive (JAR) files, a package file format used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file to distribute application software or libraries on the Java platform. JAR files contain the class bytecodes, with each class having an unique name. The Class Loader searches through these libraries and tries to find the classes contained in them.

2.4 ROM Classes and RAM Classes

Once the Class Loader finds a Java class, it is transformed into *ROM* and *RAM* Classes. The transformation is done based on the mutability of the information in the class file. If the information is constant for each version of the class, then it is stored as a ROM class and if not, it is created when each instance of the class is loaded and it is called a RAM Class. ROM classes

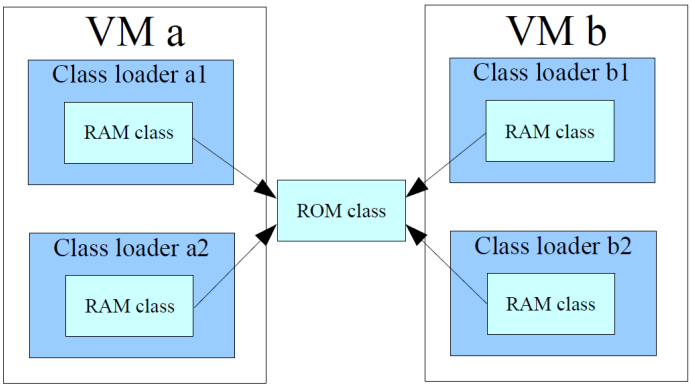

Figure 2.2: Shared ROM Class[6].

contain data like field shape descriptions, method bytecodes, annotation data and UTF8 strings.

A new RAM class is generated for every class and points to its respective ROM class. The RAM class additionally contains mutable data, which can be different in each instance, such as class specific data like class variables. Figure 2.2 depicts how the ROM class is shared among the Class Loaders.

2.5 Ahead-of-Time Compiled Code

Before Java 6, a method was executed either by interpretation of the Java bytecode that composed a method or by executing native code that is compiled and optimised by the JIT compiler. With Java 6, *Ahead-of-Time* (AOT) code compilation was introduced. It is similar to JIT code, but compilation is done before execution and the optimisation is minimal in comparison to that done by a JIT compiler. The AOT compiled code was introduced to

provide a faster startup by providing pre-compiled versions of methods [9]. It is quicker to load the AOT code from the cache for the native code version of a Java method than generating JIT compiled code, but it is not as optimised. AOT compiled code is generated before execution, hence not degrading any performance [9].

Even though AOT compiled code is stored in the SCC, it is not shared among VMs. It is copied for each VM using it. Hence, there is no direct improvement in the memory footprint but there are CPU savings from being able to reuse this code rather than re-compiling.

2.6 The Heap

The JVM has a runtime data area called the *heap*, from which memory for all objects and arrays is allocated. It is created during JVM start-up and all the objects that are created during run-time are allocated on the heap. The heap is shared among all JVM threads. The heap may be of a fixed size or may be expanded as required during execution and may also be contracted if a larger heap becomes unnecessary. In general, the memory for the heap is contiguous, although non-contiguous portions inside the contiguous space are used. A JVM implementation may provide the user control over the initial size of the heap, if the heap can be dynamically expanded or contracted, as well as, control over the maximum and minimum heap size [2].

According to the JVM specification, the heap size may be configured with

the following VM options:

- -Xmx<size>: to set the maximum Java heap size.
- -Xms<size>: to set the initial Java heap size.

If not specified through the command line, the heap size is initialised to a default size, with the ability to be expanded as required.

2.7 Shared Class Cache

With Java 5 in 2005, IBM introduced the *Shared Class Cache*, a shared memory space among all the instances of the JVM. Any information that can be used for further executions by a JVM (that is, immutable data) can be stored in the SCC. A SCC is an area of fixed size, which persists beyond the lifetime of any JVM using it. Any number of Shared Class Caches can exist on a system; however, a single JVM can only connect to one cache during its lifetime. No JVM owns the cache, any number of JVMs can read and write to the same cache [12].

The introduction of the SCC was a huge leap forward in terms of improving startup time as well as reducing the memory footprint for a VM. It provided a transparent mode of sharing data among multiple VMs. It has to be enabled to read or write information to it. Earlier JVM versions attempted some form of class sharing between multiple JVMs; for example, the IBM Persistent

Reusable JVM on z/OS, Oracle Corporation *CDS* feature in their Java 5.0 release, and the bytecode verification cache in the i5/OS Classic VM.

In the absence of the SCC, each JVM has a copy of the class file. So a chunk of memory is wasted and the startup time is higher as the classes need to be loaded from the disk every time [12]. The basic goal behind the introduction of the SCC was to reduce the virtual memory footprint. SCC is either a memory-mapped file or an area of shared memory, which has a fixed size during execution and, based on the implementation, 16MB is the default and the technical limit is 2GB. Customers use much bigger caches than 16 MB, i.e. Liberty [7] uses about 80MB by default. Once the size of the SCC is set, it cannot be expanded. Hence, once the SCC is full, a VM can only read from it but no more data can be added to it. The SCC can be of two types : *persistent* caches, which exist even after the operating system is shut down and *non-persistent* caches, which are lost when the operating system is shut down [12]. Figure 2.3 shows how a JVM searches for a class file. JVM 1 is looking for a class "C1". It first searches in the SCC and as C1 is being loaded for the first time it will not find it in the SCC. So instead, the class file has to be loaded from the file-system. Then, JVM 1 will save the class in the SCC for future use. After some time, JVM 2 tries to load the same class C1 and finds it in the cache, hence, saving execution time and memory footprint [12].

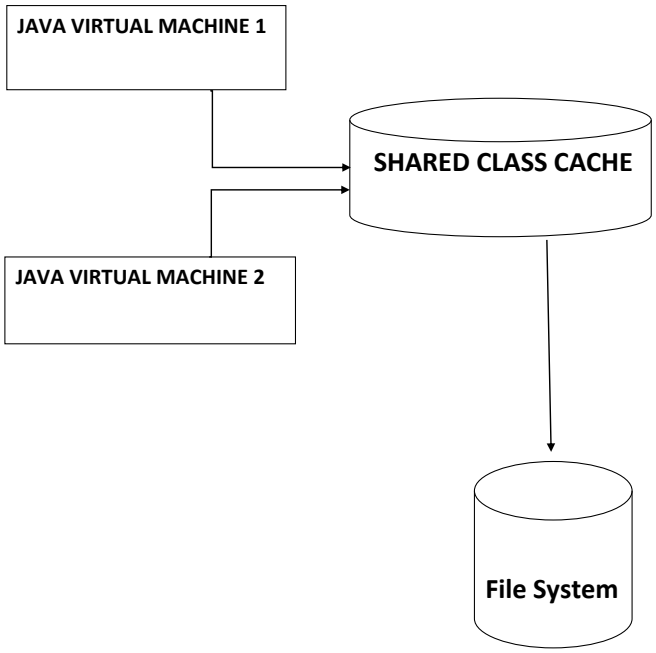

Figure 2.3: The Shared Class Cache Mechanism

2.7.1 Using the Shared Class Cache

The command "`-Xshareclasses:help`" provides all the options that relate to the Shared Cache. The Shared Class Cache can be enabled using the command "`-Xshareclasses`" and the name of the cache can be provided using "`-Xshareclasses:name= <name of the cache>`". If no name is provided, then a default name is used.

To specify the size for the cache the utility "`-Xscmx<size>[k| m| g]`" is used. The default size is 16MB. To list all the caches that were created, the

command `-Xshareclasses:listAllCaches` is used. To provide the statistics regarding a cache, the utility `-printStats` has to be specified with name of the cache. Finally, to delete a cache, `(-Xshareclasses:name=<name of the cache>,destroy)` is used, where the name of the cache to be destroyed is mandatory.

To receive feedback on how a cache is performing, the option `-Xshareclasses:verbose` is used. It reports how many bytes have been stored and read. It also reports the size of the Cache [12].

2.7.2 Information stored in the Shared Class Cache

With the introduction of the SCC, there was a greater opportunity to find immutable data and to reduce memory footprint and startup time, that was being used to do the same task by different JVMs and even Class Loaders. As the ROM classes contain immutable information, they were ideal to be stored in the Cache. So, a Class Loader can reuse the ROM class from the SCC (if enabled). If it is not found then it is loaded from the file-system and added to the Cache.

As the SCC persists beyond the lifetime of any JVM connected to it, a difficulty may arise when the class that is already loaded in the SCC is altered in the file-system. It is the duty of the SCC to solve this problem and always provide the correct version of the class to the Class Loader. To overcome this problem, there are time-stamp values stored in the SCC; when a Class Loader loads the class, it checks the class's time-stamp with the

time-stamp in the file. If it detects that the time-stamp for a JAR has been changed then it marks each class in that JAR as stale as it has no way to find the particular file. When the classes from that JAR are loaded from their actual file-system and re-added to the SCC, only the ones that have actually changed are added. Those that have not changed are kept active [12].

Besides ROM classes, the Ahead-of-Time (AOT) compiled code is also stored in the SCC. To support AOT compilation, pre-compiled methods are stored in the SCC between runs of an application. The minimum and maximum amount of the SCC that can be occupied by AOT code can be defined using command line options. If a maximum amount is not specified, then the default setting is to use the entire cache. This will not result in the entire cache being filled with AOT code because AOT code can be generated only from classes already in the cache (as shown in Figure 2.4). The pre-compiled code is stored in the SCC and when a VM tries to execute a method it copies the pre-compiled code from the SCC. As AOT code is stored in the Shared Cache, it is available not only for the current VM that stores the code but also for other VMs that use the Cache and hence reducing startup time.

Apart from the ROM classes and AOT compiled code, the SCC also stores hints for JIT compilation and JAR file indices.

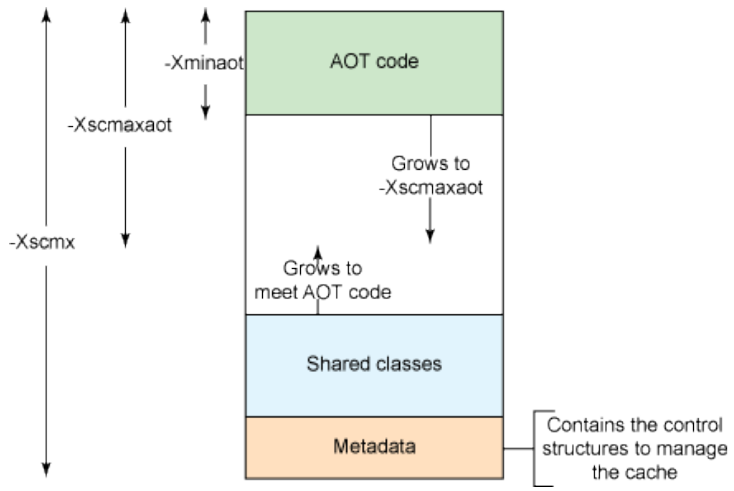

Figure 2.4: A conceptual explanation to store AOT Compiled Code [12].

2.8 Garbage Collection

"Garbage Collection (GC) is the automatic reclamation of computer storage" [14]. In a non-GC language, like C, allocating and deallocating memory is a manual process and the developer is responsible for reclaiming the space of unused variables and dead objects. In Java, the process of deallocating memory is handled automatically by the garbage collector. Java objects can be in one of two states: alive or dead. All objects that are reachable from Java threads, and other root sources are marked as alive, as well as the objects that are referenced by other live objects. Subsequently, all other objects can be considered dead. The garbage collector's job is to find objects

during runtime that are dead and free their space so that new objects can occupy their memory space. When the heap becomes full, meaning when an object cannot be allocated, garbage collection begins, but there might also be other reasons for calling a GC such as a high water mark.

2.8.1 GC policies

The heap can be one whole entity or may be divided into smaller sections depending on the GC policy. In this section a brief introduction of the various GC policies used in J9VM is provided. Each GC policy performs the same task of reclaiming dead objects, but they differ in the way they operate.

Tracing garbage collection is the most common type of garbage collection. The overall strategy consists of determining which objects should be garbage collected by tracing which objects are reachable by a chain of references from certain root objects, and considering the rest as garbage and collecting them. Reference counting is a form of garbage collection whereby each object has a count of the number of references to it. Garbage is identified by having a reference count of zero. An object's reference count is incremented when a reference to it is created, and decremented when a reference is destroyed. Variants of tracing include mark-and-sweep, mark-and-compact, and the copying GC [1].

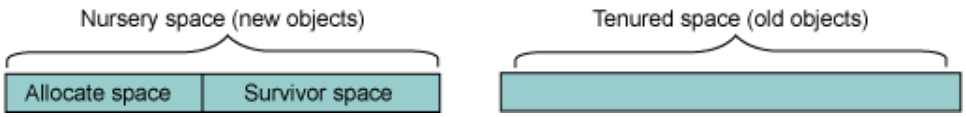

Figure 2.5: Heap layout for GenCon [13]

2.8.2 Generational Concurrent Garbage Collection

The generational concurrent garbage collection, also known as "GenCon", is the default GC policy for the J9VM. It follows the principle of "objects die young"—that is, objects do not survive many garbage collections. GenCon considers the lifetime of the objects and places them in separate areas of the heap. The heap is split into two areas—nursery and tenure space [13]. Objects are created in the nursery and, if they live long enough, are promoted into the tenured area. Objects are promoted after having survived a certain number of garbage collections. As most objects are short-lived, collecting the nursery frequently frees these objects without paying the cost of collecting the entire heap. The tenured area is garbage collected less often.

Figure 2.5 shows the layout for a heap using GenCon. The figure shows how the nursery space is split into allocate space and survivor space. Objects are allocated into the allocate space and, when that fills up, live objects are copied into the survivor space or into the tenured space, depending on their age. The spaces in the nursery then switch use, with allocate becoming survivor and survivor becoming allocate. The space occupied by dead objects

can simply be overwritten by new allocations. Nursery collection is called a scavenge. The policy that is used to manipulate the nursery is called copying collection. Each object stores its age, i.e. the number of GC cycles it survived to date. All objects alive at this point, with an age above a certain threshold, are copied into the tenured space, the remaining live objects are moved into survivor space. The threshold can be set to a fixed value but is usually automatically adjusted by the VM based on memory usage and the average life expectancy of objects. Figure 2.6 depicts how the heap looks before and after a GC is done.

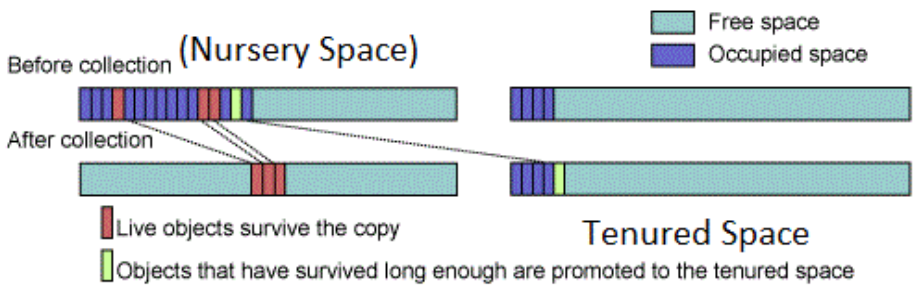

Figure 2.6: Heap layout before and after a GC using GenCon policy [13].

The GenCon policy has a concurrent aspect to it. The tenured space is concurrently marked but without a concurrent sweep. All allocations pay a small throughput tax during the concurrent phase. With this approach, the pause time incurred from the tenure space collections is kept small [13].
Figure 2.7 shows how the mutator and the GC threads work concurrently. A mutator thread is the application program that allocates objects. A scavenge

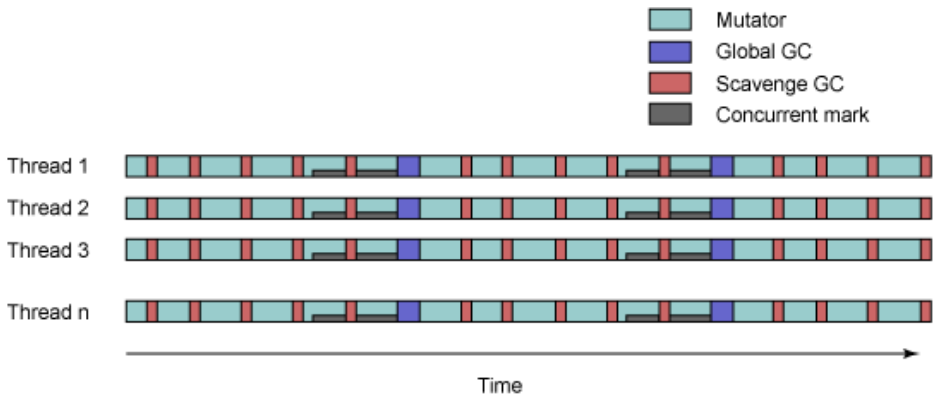

Figure 2.7: Distribution of CPU time between mutators and GC threads in GenCon [13].

run is short (shown by the small red boxes). Gray indicates that concurrent tracing starts followed by a collection of the tenured space, some of which happens concurrently. This is called a *global collection*, and it includes both a scavenge and a tenure space collection.

2.8.3 Optimal Throughput Garbage Collection Policy

Before concurrent garbage collections were used, J9 VM's main policy was the Optimal Throughput policy. It consists of a mark and sweep phase and a compaction phase, which could be used under special circumstances such as fragmentation. During the mark phase, starting from JVM internal root objects, the whole object graph is traversed and all visited objects are marked as alive. The heap is then traversed a second time, removing all objects that have not been marked as alive [21]. If at the end of a garbage collection run,

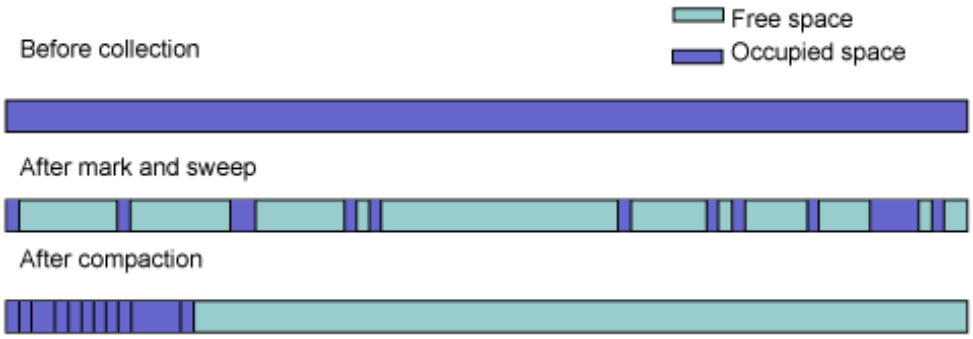

Figure 2.8: Heap layout after compaction [13].

there are a number of small memory fragments, then compaction is done to reduce fragmentation, as seen in Figure 2.8.

Compared to policies with a segmented heap like generational concurrent, this approach has some disadvantages which eventually led to its replacement as the default policy. Collection overhead grows linearly with the used memory, as the whole space has to be traversed for each collection cycle. As this policy needs to stop all other executing Java threads, long halts are introduced, especially when used with large heap sizes. Copying mechanisms only need to traverse the memory once, directly moving the data. Here, the heap has to be traversed twice for marking and moving. Removing objects also lead to heap fragmentation in the long run. If this happens a compaction step is necessary to realign the memory. This operation can last a relatively long time, not allowing any other operations to run while being executed [13].

As no garbage collection related operations are run during normal execution, this policy excels at delivering high performance and throughput for short periods of time. Once the garbage collection kicks in, this advantage in performance is negated as collection operations take up more time. Especially for long running applications where frequent compactions are necessary, this performance impact is noticeable.

2.8.4 Balanced Garbage Collection

The balanced garbage collection policy was introduced in the J9 VM with the goal of reducing performance fluctuation during garbage collection and to even out the average performance as depicted in Figure 2.9, with GenCon being the "current" policy. The policy uses a hybrid approach to garbage collection by targeting areas of the heap with the best return on investment. The policy uses mark, sweep, compact and generational style garbage collection [10]. The primary goal of the balanced collector is to reduce the cost of

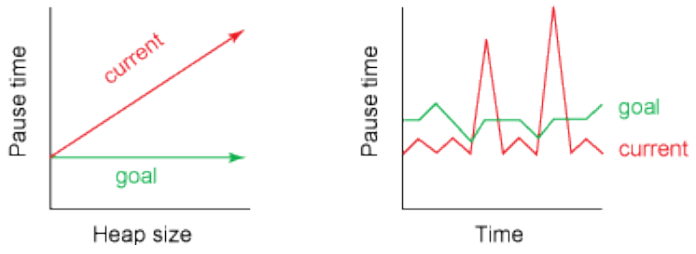

Figure 2.9: Pause time goals of the balanced collector with GenCon being the "current policy" [10].

global garbage collection, reducing the effect of whole heap collection times. At the same time, each pause should attempt to perform a self-contained collection, returning free memory back to the application for immediate reuse [10]. As the heaps expands, it means a larger area has to be covered during garbage collection, which results in longer collection pauses.

Reasons for elongated collection pauses are as follows:

1. "If the size of the live data grows, it will take longer to iterate over all live objects."
2. "If a copying algorithm is not used, fragmentation occurs. This has to be handled, further imposing performance hits on the Java runtime."
3. "If the application has an unusually high rate of new object allocation, the rate of garbage collection will rise as well, dampening performance. Size and longevity of the objects can also play a role at this point." [10]

The balanced garbage collection policy divides the Java heap into smaller regions, each of which can be collected individually. These regions are individually collected to reduce the maximum pause time on large heaps, and also benefit from Non-Uniform Memory Architecture (NUMA) characteristics on modern server hardware [16]. The choice of which regions will be garbage collected is made at the start of each new garbage collection cycle and depends on the heap usage pattern of the application [10]. The balanced garbage collection policy is intended for environments where heap sizes are greater than 4 GB. The policy is available only on 64-bit platforms and can

be activated by specifying `-Xgcpolicy:balanced` on the command line.

The size of the regions is decided during the JVM startup and can not be changed thereafter. Growing and shrinking the heap equals adding and removing regions. Generally, the size of a region is determined as Xmx/1024, rounded down to the nearest power of 2, where Xmx is the maximum heap size available. However, a region cannot be smaller than 512KB and there must be fewer than 2048 regions, so the calculation is adjusted accordingly. Except for small heap sizes this typically means between 1024 and 2048 regions are selected.

For the Garbage Collector, each region is self contained which means that all operations like tracing (determining the relationship between objects) and compacting will happen inside a single region. The region an object is assigned to is chosen depending on the objects characteristics. Similar objects are grouped together. This can for example mean that objects of similar sizes or age are put next to each other in one region. This also means that not only objects but regions as well are assigned an age which can range from 0 to 24 [10][8]. Regions with the age of zero contain newly created objects and are called Eden regions. The whole layout of the region-based heap can also be seen in Figure 2.10.

To make use of the principle that newly created objects are more likely to die soon, eden regions are collected each cycle, called the *Partial Garbage Collection* ("PGC"), which halts the VM execution until the garbage collection is finished. The collection might also include regions other than the

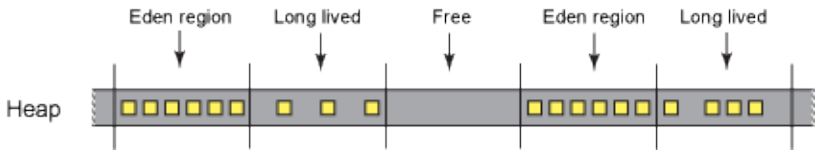

Figure 2.10: Region structure and characteristics found in the object heap [10].

eden space, if the collector determines that these regions are worth collecting. When the collection is complete, the application threads can proceed, allocating from a new eden space, until this area is full. This process continues throughout the lifetime of the application. From time to time, the collector starts a *Global Mark Phase* (GMP) to look for more opportunities to reclaim memory. Because PGC operations see only subsets of the heap during each collection, abandoned objects might remain in the heap. However, the GMP runs on the entire Java heap and can identify objects that are dead so that they can be reclaimed.

2.8.4.1 Partial Garbage Collection (PGC)

Partial garbage collection is triggered when, for example, the eden space is running out of memory. During the collection, fragmentation issues are also resolved. When a PGC is run, regions are added to the collection set, and are scheduled to be collected. Whether a region is added to a collection set

or not is dependent on the following factors:

1. "Eden regions are always added to the collection set due to the likelihood of new objects dying very soon."

2. "Heavily fragmented regions are added to the collection set as freed memory is needed for future eden allocations."

3. "Non-eden regions that contain objects that are likely to die during the next cycle are added to the collection set." [10]

Similar to the generational concurrent collection policy, a copying approach is used to counter fragmentation. As there is no fixed region which can serve as a target for the copy, it is necessary to reserve multiple free regions as targets. The exact number is again adapted dynamically during runtime, mostly based on expected mortality rates of collection set regions. If the computed values repeatedly prove to be wrong, another mechanism which is capable of tracing and compacting a region without using any additional memory can be activated. A hybrid approach where some objects are copied and others are handled in place is also possible. PGC operations are of a stop-the-world nature, halting all other Java threads until the collection cycle is finished.

2.8.4.2 Global Marking Phase (GMP)

Parts of the global marking phase run concurrently with the allocation of objects. GMP is used to trace and mark objects and determine their state. The data collected during this phase is also used to decide which collection

operation is appropriate for a specific region. Since marking all objects can take some time, a GMP can be executed iteratively while multiple collection cycles occurs. While the PGC is capable of freeing large amounts of memory, it has the shortcoming of only operating locally on a specific region. As PGC deals with only certain regions and not the whole heap, this makes the decision of which region to add to the collection set unreliable over time. The GMP provides region spanning information, massively improving the capability of choosing an appropriate region. In case the PGC becomes too inefficient or its efficiency drastically drops, a GMP is executed to gain information improving future collections.

2.8.4.3 Global Garbage Collection (GGC)

If the VM runs into very tight memory conditions it can request a global garbage collection. During a GGC, all other Java threads are stopped and a mark, sweep and compact operation is performed on the whole heap. As this is a very costly operation, the VM tries to avoid a GGC as long as possible. The interaction of the different collection phases when using a balanced garbage collection policy ensures that there are no overly long halts during a VMs execution. Figure 2.11 gives an example on how the different phases work together.

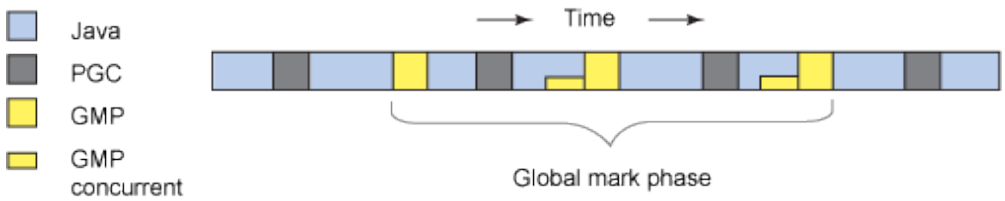

Figure 2.11: Example of a balanced GC run [10].

2.9 Benchmarking

Benchmarks can be used to measure the improvement in performance that was obtained by changing the original code. These benchmarks provide the developer with a way to decide if the altered code actually improves the performance or not. The developer can compare each individual result against the original implementation and decide if it is useful to include these changes into the existing systems.

In this project, benchmarking was done to compare the performance and memory footprint of an application with the SCC containing no new changes and the SCC containing the newly added information.

For this research the benchmarks that were used are:

1. The DaCapo benchmark suite [4].

2. Liberty Daytrader. DayTrader is an Open Source benchmark application emulating an online stock trading system [11].

The following sub-benchmarks in the DaCapo suite are used in our experi-

ments:

- batik: A Scalable Vector Graphics (SVG) images tool based on the unit tests in Apache Batik.
- fop: An XSL-FO file parser and formatter.
- jython: A Python interpreter.
- luindex: A text index tool using Apache Lucene which is a text.
- lusearch: A text search tool using Apache Lucene.
- pmd: A Java classes analyzer.
- sunflow: An image render using ray tracing.
- tomcat: A web service simulator using Tomcat.

WebSphere Liberty is a fast, dynamic, easy-to-use Java EE application server. Liberty is a combination of IBM technology and open source software, with fast startup times (<2 seconds), no server restarts to pick up changes, and a simple XML configuration. Liberty has two applications, Tradelite and Daytrader. Liberty is a lightweight subset of WebSphere Application Server (WAS)[7], a framework for making server applications. Daytrader is one server application that is built with Liberty, and TradeLite is a more lightweight version of it. Both Daytrader and Tradelite may be used to measure the startup time and memory footprint for starting the server. Apart from

start-up and memory footprint, the Liberty benchmark was setup to provide throughput.

2.10 Related Work

Research has been done in this project to understand the work that has been performed in the field of using a Shared Class Cache. Wong et al [19] proposed that the JVM is slow when starting up, as hundreds of classes need to be loaded. To solve the problem they added the feature of code sharing and the storage of the constant pool. Even though this approach is similar to what is done in the J9VM, they do not store any further information. Similarly, Kawachiya et al [15] propose a way to make the JVM faster by copying an already initialized Virtual Machine and starting the new instances of the Virtual Machine from there. They saved the time of starting up from scratch, but the main disadvantage of this proposal is that they had the same configuration for all the Virtual Machines, which might not be desirable. Plus, their proposal is not implementable as no Java VM is cloneable. Czajkowski et al [5] suggest a similar idea of sharing code among instances of the Virtual Machine. Neu et al [18] demonstrate that fine-tuning additional JVM parameters could produce significant performance improvements. Specifically, they conducted experiments setting the heap size, the nursery space size and the tenuring rate to fixed values before running the VM.

This work is different from the above as it includes storing additional infor-

mation in the SCC beyond the class pool. Some of the information can be fine-tuning of JVM parameters as suggested, but not implemented, by Neu et al. However, our work goes further by storing additional low-level internal information that can be used by a JVM to achieve better performance.

Chapter 3

Project Design

This chapter defines the design of this research, starting by stating the problem in section 3.1 and the basic requirements in section 3.2. The approach for this project is explained in Section 3.3. Finally, the development environment is presented in Section 3.4.

3.1 Problem Statement

The JVM's start-up is time-consuming and others have researched to find solutions for this problem [15][19]. The purpose of this thesis is to find additional data that can be stored in the SCC, to improve the start-up time and also to reduce the memory footprint.

There is substantial information that is already stored in the SCC. The idea is to find more immutable data to store in the cache so that the JVM does not

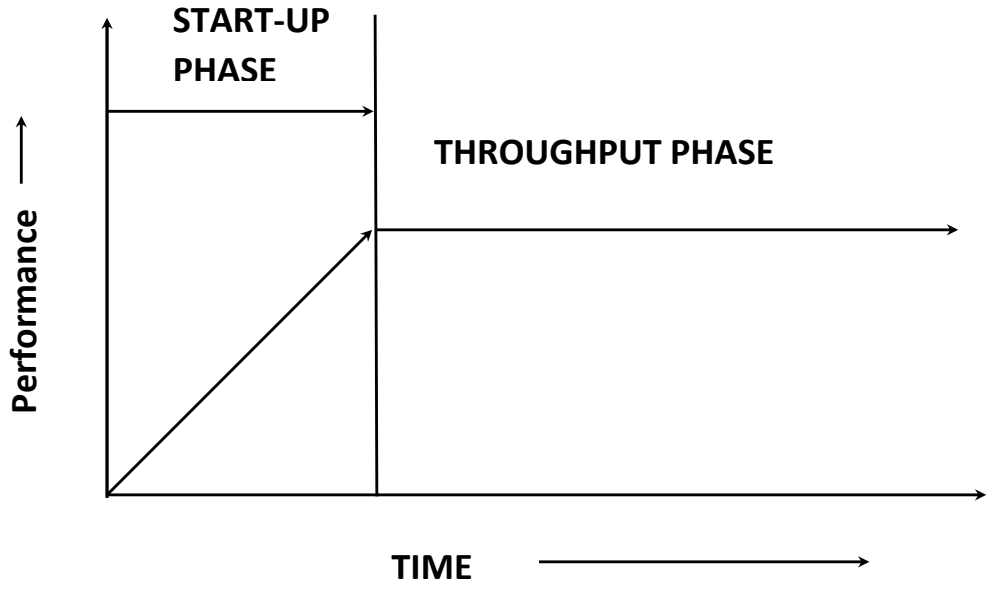

Figure 3.1: Startup Phase

have to do the same task repeatedly and instead can just search in the cache and reuse the previously stored data. Improving the start-up speed will lead to quicker execution of an application; hence, starting the throughput phase as soon as possible. The throughput phase is where an application attains maximum performance. Figure 3.1 shows how an application's life cycle can be divided into the two different phases i.e. start-up and throughput.

During the start-up phase, the JVM performs all the necessary tasks that are needed to run an application. The JVM performs these tasks in a sequential order, an order which, if changed, may stop the JVM from running. The JVM starts by loading all the default libraries, including C libraries

that are required for starting an application. Then, the JVM initializes the heap. Once the heap structures are initialized, the system Class Loaders are initialized. Finally, the JVM loads the SCC and the JVM reaches a stage where all default initialization is complete. On successful completion of these phases and if there is no application-specific initialization, the JVM can run an application and this is when the application attains maximum performance.

As all the phases are essential, the only way to achieve improvement would be to reduce the time the JVM spends on doing repetitive tasks. If a JVM performs a particular task more than once and if the data is immutable, then the information could be stored in the SCC for future runs. The following chapters will give detailed insight on the data that was added to the SCC. Finally, these changes were evaluated to see if any performance improvement were attributed due to them.

3.2 Requirements

The following are the basic requirements for the solution:

1. The new approach should be easily understandable by every programmer using the JVM.

2. The alterations made to the JVM should not change the JVM as well as any user program drastically. This means that the way the programmer

uses the SCC does not change or the programmer does not have to manually store any information in the SCC.

3. The JVM should work as before if the new information stored in the SCC is removed. There should be no changes to the functionality of the JVM and the changes must be kept minimal, so that as few files as possible need to be altered. When changes are made, these changes need to be evaluated to determine if they are working correctly with and without the new implementation being active.

4. The new information stored in the SCC should at least keep the start-up time and memory footprint the same as before and not hamper the performance.

3.3 Approach

In order to fulfill the requirements stated in Section 3.2, the entire project was sub-divided into smaller tasks. The task of finding any new information is performed by looking into the start-up phase and checking if there is immutable data that is used multiple times. After the initial analysis, there were multiple areas in the JVM that could be stored in the SCC such as:

Case 1: Size of the buffers used by the JVM during start-up.

Case 2: Heap size and information related to garbage collection policies.

Case 3: Setting the size of the Class Loader block and the size of the thread pool.

During class loading, the JVM uses multiple buffers to store information, and the size of these buffers is expanded on multiple occasions. The final buffer sizes, if reused from the SCC, could be beneficial. Similarly, information related to heap sizes was another aspect that proved to be very effective for this research. The heap sizes were specific to the Garbage Collection policy in use. Garbage Collection can be time-consuming, more so if it is a stop-the-world Garbage Collection, and also if the heap is set to a smaller size and is expanded on several occasions. Hence, reducing the number of Garbage Collections will significantly improve the execution time. Other data stored in the SCC is discussed in the coming chapters.

3.4 Development Environment

As the JVM runs on various hardware and software configurations, it was necessary to select a setup that will be used for this research project. As the University of New Brunswick (UNB) and IBM supplied their Centre for Advanced Studies (CAS) laboratory with computers, these will be used for implementing this project. The machine used for this project consists of an Intel Core i7-2600 processor, which consists of four cores and due to its hyperthreading ability it can utilize eight threads simultaneously. Furthermore, this computer features eight gigabytes of RAM. Benchmarking was performed on a server with a 1.8 GHz 16 core/32 Thread Xeon, 4x E7520, Nehalem-based server. This server is used for compiling as well as executing

the entire project.

The specified hardware features an Intel x86 compatible CPU, therefore, only software compatible to this architecture is used. With the Microsoft Windows operating system being one of the most widely used operating systems for consumers, the decision has been made to use Microsoft Windows 7 Professional (64 bit version). The Eclipse IDE will be used for this project as well.

The IBM JVM is written in the C and C++ programming languages, as well as in the IBM proprietary Builder programming language. These three languages are supported by the Eclipse IDE either natively or with plugins and therefore, these languages will also be used for the implementation of this project.

Chapter 4

Implementation

This chapter describes the implementation. There are three main aspects of the JVM that were examined: buffers, the heap size and setting the size of the thread pool and Class Loader blocks. This chapter explains what was already implemented before this project started and what changes were made to the code. In Section 4.1, the changes made to several buffer sizes in the JVM and why it might be influential in terms of lowering start-up time is explained. Section 4.2, explains the changes made to the heap size and the different Garbage Collection policies that were affected by it. Finally Section 4.3, explains the changes made to the pool sizes.

4.1 Buffer Implementation

Throughout the start-up phase, the JVM uses multiple buffers to store temporary data. It can store the size of the StackMap table, the name of a class or sometimes the JVM uses buffers during the transformation of ROM Classes. A *StackMapTable* consists of zero or more stack map frames. Each stack map frame specifies (either explicitly or implicitly) a bytecode offset, the verification types for the local variables, and the verification types for the operand stack. Every time a buffer is used, its size is checked to ensure that it is big enough to hold the data; if not, the buffer is reallocated. By default these data structures start small and expand on demand, as execution warrants. Starting with large data structures is wasteful of the available resources and can consume unnecessary execution time during initialization if the resource is not needed; starting small and growing these structures can lead to repeated allocation and data copying, which affect performance and fragment memory.

We observed that if the same application is run again, it is likely that these buffers will grow from their initial values to the same maximum values. Thus, it would be more efficient if the application could start with the same maximum buffer size as used during its previous run. As these buffers are used every time a class is loaded, it is better to find a more economical way than reallocating the buffer every time. During the life cycle of this project, several buffer sizes were altered to reduce the execution time.

4.1.1 Maximum Stack Map Buffer

During the creation of RAM classes for the corresponding ROM classes, the size of the global buffer *mapMemoryResultsBuffer* is calculated. The size is calculated to ensure there is enough space in the buffer for that particular ROM class. The buffer is used when walking a thread's Java stack to determine which stack slots hold live objects. The JVM prefers a small stack-allocated buffer, but falls back to this global buffer if there are more than 32 local stacks for a method and if malloc returns null when attempting to allocate a buffer.

The size is calculated for every method in a class, by iterating over the methods in the ROM class, to calculate the maximum stack map size required. The global buffer is reallocated if the calculated stack map size is more than the default global buffer size.

The reallocation is done every time the newly calculated maximum size of the stack map is greater than the default size of the buffer. In this project, the calculation of the maximum stack map size is done differently. At the end of the first run of an application, the maximum size of the buffer is stored in the SCC. For every future run, before initializing the buffer, the SCC is checked for the previous run's maximum buffer size and if there is a value stored then the buffer is set to that size. For future use of that buffer, it is reinitialized only when the currently calculated buffer size is greater than the cached value. Finally, if the buffer is reinitialized then the size stored in SCC is updated to reflect the maximum size of the buffer. Hence,

the buffer is initialized to the maximum stack map size stored in the SCC and reinitialized only if there is a new maximum. So, the improvement in performance is achieved due to the fact that there is fewer reallocations of the buffer. Thus, with the new implementation, execution is faster. Chapter 5 provides a detailed description of the improvement achieved.

4.1.2 Buffers used during Dynamic Loading of Classes

A class represents the code to be executed, whereas data is the state associated with that code. The JVM is responsible for loading and executing code on the Java platform. It uses a Class Loader to load Java classes into the Java runtime environment.

During the class loading process, the following buffer comes into play. It is used in dynamic loading of a class. The buffer stores the name of the class that is being searched for. Thus, the buffer size has to be checked every time to see if it is big enough to store the class name. If it is not, then the buffer is expanded. The check is done using a macro which, is used on multiple occasions during this phase, each time with a different kind of class, but the functionality is the same. If the current size is insufficient, the current buffer is freed and then reallocated.

Instead of expanding the buffer every time, we now store the maximum size of the buffer from the previous run in the SCC. During future runs, the buffer is allocated to the maximum size stored in the SCC. Figure 4.1 shows how the new implementation calculates the size of the buffer.

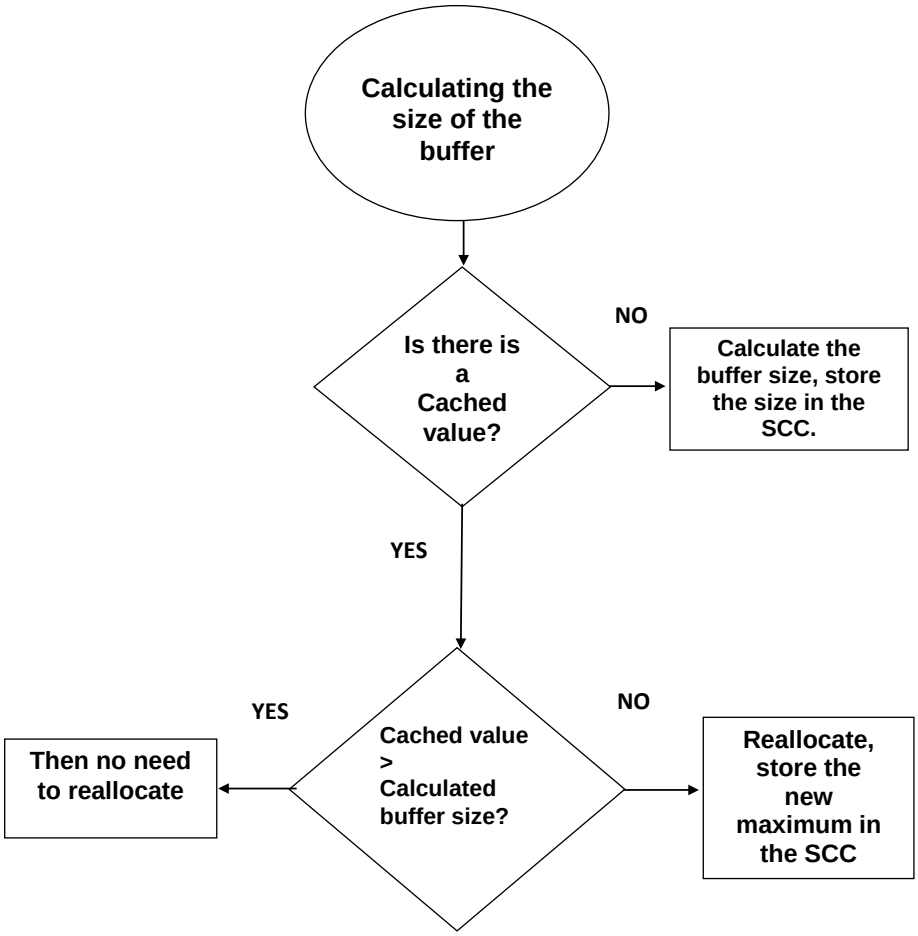

Figure 4.1: Modified Buffer Size Calculation

4.1.3 Buffers used during ROM Class Creation

When a Class Loader successfully loads a class for the first time, the class bytecode is translated into ROM and RAM classes, based on the characteristics of the information.

During the ROM class building, the bytecode is first analysed in the *class*

file parsing phase and then the translation of the class bytecode into ROM class takes place in the *prepare and laydown* phase. During the prepare and laydown phase, all the data structures, like the constant pool and a table for all the pointers used in that class, are created, but first the SCC (if enabled) is checked for the ROM class. If a ROM class identical to the one being created is already present then the ROM class from the cache is returned. During these phases, two buffers are used to store information that is related to building a ROM class; one in class file parsing and the other in prepare and laydown phase.

4.2 Varying the Heap Size

If not specified through the command line, the heap starts at a default size. As objects are allocated in the heap, the heap would become full. When an object cannot be allocated, a garbage collection is triggered and might result in the expansion of the heap and the application would continue its course. The heap would be increased on several occasions to meet the needs of the application. The amount of execution time spent on garbage collecting as well as expanding the heap is considerable. For some GC policies like OptThruput, which does not divide the heap based on age, the garbage collection overhead grows linearly with the used memory, as the entire heap is traversed in order to find dead objects.

In this research, during the first run of an application, the maximum heap

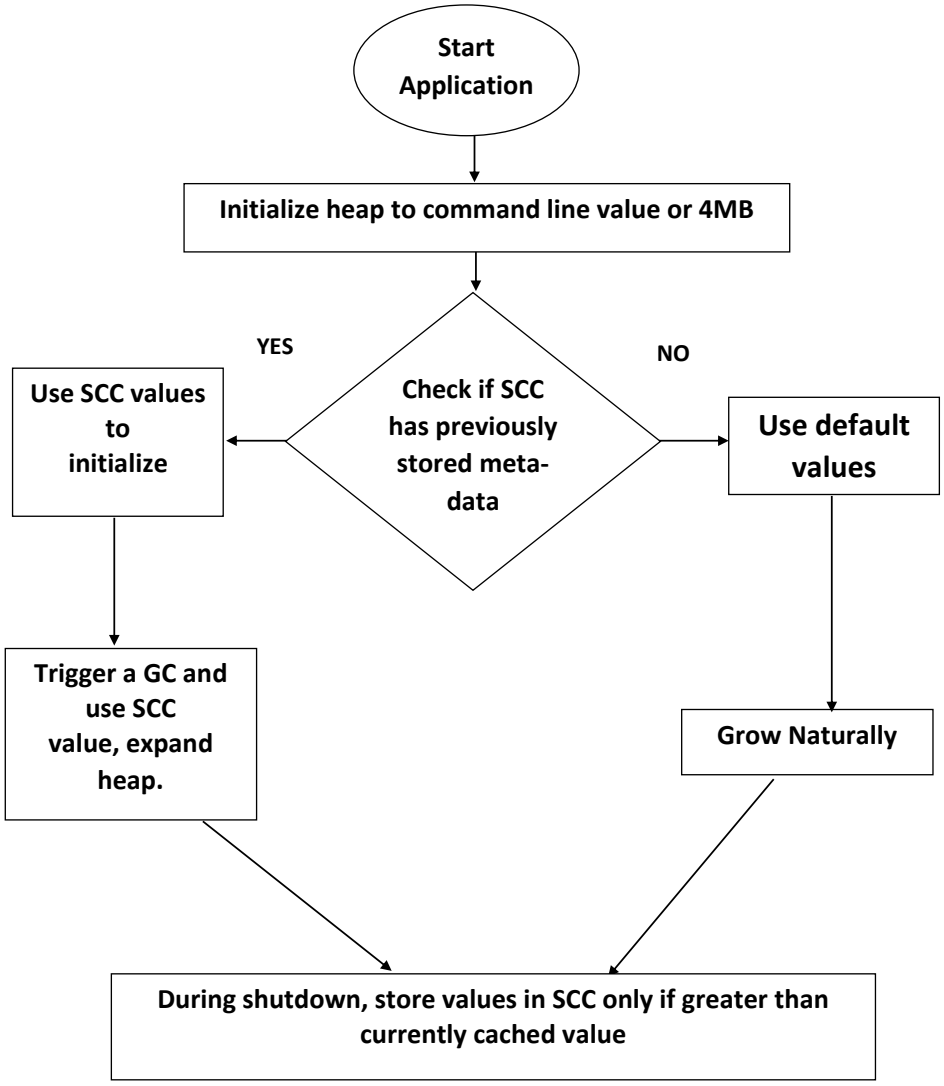

Figure 4.2: Flow of Extending the Heap

size used during that run is stored in the SCC. The heap size is stored in two different occasions, just to assess which gives the best performance. One was after the start-up phase was finished and, the second is stored during the shutdown phase. The heap size after the startup time was almost 1/3 of the maximum heap that the application uses. Hence, the number of GCs were increasing, when the heap size after start-up phase is used. Thus, leading to a slower execution time in comparison to when the heap size was stored during shutdown, which is often the maximum heap used by the application. It was trickier to actually realize when exactly the start-up phase for a JVM is finished. The JVM does not have any kind of notifications in order to state that the application is finally in its throughput phase. So, in order to retrieve the heap size when the start-up phase is over the JIT was used. The JIT has a trigger that specifies when the start-up phase is over and during that phase the heap size was retrieved.

Once the heap size of the application is stored in the SCC, during future runs, the heap size is retrieved from the SCC and the current heap size is changed to the size stored in the SCC. Initially, the heap is altered to the desired size during the first GC, but even then the application has to wait for the heap to be full. A substantial performance gain was observed, but the gain could be increased if the heap size could be altered to the size stored in the SCC as soon as possible. There were two reasons why the heap could not start with the previously stored heap size from the SCC. The JVM start-up happens in a fixed order and the order is that the heap structures are initialised, which

includes the initialisation of the heap, then the class loaders are initialised and after that the JIT and the Shared Class Cache are initialised. Thus, the heap size could not be retrieved at the start as the SCC is initialised after the heap size is set. The second reason was that the heap can only be expanded during a GC. To perform the resizing of the heap without calling a GC, was tried but the method that is responsible for resizing needs an environment that is only available during garbage collection. Thus, the heap size is increased during the first GC. To maximize the performance, as soon as the SCC is active, a GC is triggered explicitly such that the heap expansion can occur much before the total heap is full. At the point of time during JVM initialization when the SCC is active, the number of objects allocated in the heap is not substantial. Hence, when the GC is triggered the collector does not spend a considerable amount of time except to expand the heap.

In case of the OptThruput GC policy, there is only one heap size that is stored in the SCC, which is the total size of the heap. If the GC policy is GenCon, then nursery size as well as the tenure size are stored separately. Based on the policy, the respective size is used. For GenCon it is trickier, as two values are being used for tenure and nursery and they have to be expanded simultaneously. For GenCon, the first GC call is generally a scavenge GC, which deals only with the nursery, so resizing the tenure space during the first GC was more difficult to implement. The last step in this implementation was to correctly add the reason for expansion in the log file, so that when the expansion occurs, the user can know why and what caused the expansion.

For all the heap expansions that occured during the project the reason was stored as *HEAP_EXPANSION_DUE_TO_CACHE*.

The Balanced GC policy is a new addition to IBM's JVM. For Balanced GC, the region size is fixed and the heap expansion happens only by adding new regions. In the new implementation, during the first run the total number of regions used by an application is stored in the SCC and during future runs, the total number of regions used were retrieved. Similar to the other policies, during the first GC, the heap was expanded by adding the number of regions that the application had used. This proved to be very effective as the number of garbage collections was diminished. Hence, the performance gain was substantial and are shown in Chapter 5.

In order to separate each GC policy implementation, a method is implemented (Figure 4.3). The method has three cases, one for each policy. Depending on the policy selected by the user, a different set of values are retrieved from the SCC and a different set of functionalities are performed to maximize the performance of the JVM.

4.2.1 Dynamic Prediction of the GC policy

The paper by Neu et al [18] states that based on certain criteria the GC policy can be switched from GenCon to Balanced. They did intensive research on what criteria should be set and the criteria are as follows:

1. "High average collection times can be countered by switching to the

```
switch (gcPolicy){
        Case gc_policy_optthruput:
                Retrieve the heap size from the SCC.
                break;
        Case gc_policy_gencon:
                Retrieve the nursery and tenure size from the SCC.
                break;
        Case gc_policy_balanced:
                Retrieve the number of regions from the SCC.
                break;
}
```

Figure 4.3: Implementation to Switch between Policies

balanced garbage collection policy. For GenCon, an average global collection time takes four times more than the average time spent for a partial collection for Balanced GC."

2. "If the overall time spent in the garbage collection phase is over four percent, a switch to the balanced policy might prove beneficial."

3. "Unusually long collection time for a single collection can also hint that a switch could be in order. For a segmented policy, a time of more than four seconds spent in a global collection cycle or more than two seconds spent on a partial collection can be considered overly long."[18]

Based on these criteria, information was added in the SCC so that either the policy can be changed automatically during the next run or at the end of a run, the users can be informed which policy to use. Changing the policy automatically would be tricky as the heap structures are already initialised

before the correct policy can be retrieved from the SCC but as the heap is not used before SCC is active, changing the GC policy automatically would not hamper the performance to a great extent. Benchmarking was performed in order to assess the improvement that can be gained by switching policies. The results are shown in the next chapter and it can be seen that a change in GC policy with correct arguments can be very effective.

4.3 Pool Size

During the start-up phase, one of the most important steps is to set up all the VM data structures that are used during the execution of an application. The phase is called *HEAP_STRUCTURES_INITIALIZED*, which is responsible for setting up the heap and all the other pools like the Class Loader pool and the thread pool. These pools are structures that keep information about each Class Loader or thread. Whenever a new Class Loader or a thread is created, it is added to their respective pools. Usually, these pools are stored as a set of several *puddles* with each puddle containing a specific number of items. A puddle is a data structure used to manage the memory and in case of pool structures, it stores the Class Loaders as well as the threads. It is used in the pool to store elements. So, when a puddle is full, a new puddle is added to contain the new element. Previously, each puddle had a maximum size of 20 elements. The creation of a puddle is time consuming and can hamper the performance of an application. Instead, it could be beneficial to

remember how many Class Loaders or how many threads the application used in the previous run. During this project, the exact number of Class Loaders and threads were remembered in the SCC and during future execution, the pools were set to the value stored in the SCC. In general, if an application uses more than 20 threads or Class Loaders, then this implementation will prove beneficial for such cases. Instead of waiting to increase puddles when required, it is better if the puddle size is pro-actively increased during the initialization stage.

Chapter 5

Evaluation

This chapter describes the evaluation of all the data that was added to the *Shared Class Cache* during this project. Section 5.1 explains the approach used for the evaluation whereas Section 5.2 shows the evaluation of the individual cases. Once each module has been tested and assessed, all the modules can be benchmarked together in order to determine the performance impacts of the changes that were applied.

5.1 Approach

In order to evaluate this project, the implementation first needs to be verified for correctness. During this project, several data structures were altered and each case has to be tested separately. All the changes have to be evaluated in order to ensure that the changes do not impact the functionality of the JVM.

Next, benchmarking has to be done in order to see how these changes impact the performance of the JVM. Finally, all the changes have to be cumulatively tested and benchmarked to see the overall performance impact.

A set of benchmarks was selected to evaluate the performance of our changes. Four metrics were chosen: start-up time of the JVM/ total execution time of the application; the memory footprint; change in the number of reallocations that occurred; and the overall throughput. In general terms, throughput is the rate of production or the rate at which data can be processed. To assess the throughput, the Liberty benchmarks were used. Using Liberty, 10 requests were sent concurrently and the JVM that processed the most requests per second is said to have a better throughput. In total 10000 requests were sent. The two sets of benchmarks used are the DaCapo Suite[4], which measures the total execution time of each sub-benchmark based on operations done over a specified workload and the Liberty benchmarks [7], which provide the start-up time and the memory footprint of the application.

5.2 Evaluating each Implementation

This section evaluates the implementation done during this research. Section 5.2.1 evaluates all the changes made to buffer sizes, Section 5.2.2 evaluates the changes made to the heap sizes and Section 5.2.3 evaluates the pool size changes.

5.2.1 Buffers

We stored the maximum sizes of the Maxmap buffer, used during dynamic loading of a class, and the buffers used during creation of ROM classes in the SCC, such that during future runs those buffers can be initialized to the desired size stored in the SCC. Tests were performed to check the number of times a buffer is reinitialized with the new implementation. Any increase in reinitialization leads to a slower execution time.

Each buffer size change was evaluated with several sets of benchmarks. For each benchmark, a new SCC was created and before the evaluation was done there were several cold runs such that the optimal performance of each application could be recorded.

Buffers	Unaltered JVM	Altered JVM
Maxmap	12	1
ROM Class Builder	10	2
ClassFile Parser	10	4
Dynload.c	17	4

Table 5.1: Number of Times each Buffer is Reallocated

Table 5.1 shows that for the Batik [4] benchmark there is a drastic change in the number of reallocations. Batik, from the DaCapo suite, is shown due to its heavy use of buffers and large number of allocations. As the number of reallocations decreases for the new implementation, the performance of the application also improves.

Figure 5.1 shows the performance gain that was achieved when the alterations to the Maxmap buffer was compared with the original VM. To account for possible runtime fluctuations, 50 iterations were used. The benchmarks shown in Figure 5.1 are from the DaCapo Benchmark suite. To measure the performance change, two values are presented: the average runtime of the unchanged JVM, as well as the average runtime of the altered JVM.

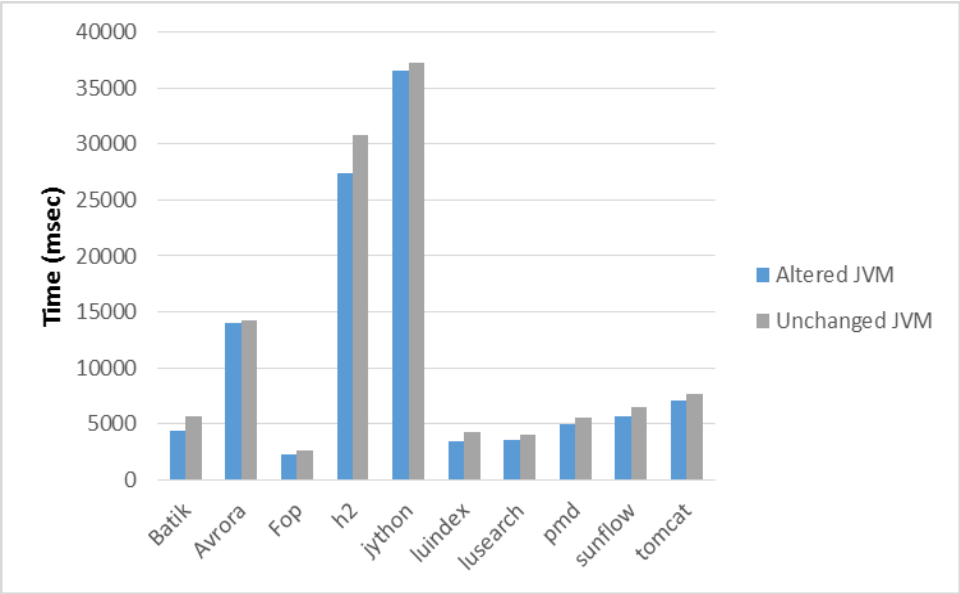

Figure 5.1: Comparing the Performance after the Maxmap Changes

The raw data of the benchmarks can be found in Table 5.2. It can be seen that when the Maxmap buffer alterations are done, for all the benchmarks except Avrora and Jython there is a minimum of 7% performance improvement. The table shows the total execution time that the application required (in msec). Improvement is observed in execution time of each benchmark,

especially with Batik and fop which show more than 15% improvement in execution time. Reallocation of the buffer by calculating the buffer size and reinitializing it is very time consuming and can be controlled by using the SCC. Results show that the buffer sizes if stored and reused from the SCC, can prevent the amount of reallocations that could have occurred.

Benchmarks	Unchanged JVM (msec)	Maxmap Buffer Implementation (msec)	Improvement (in %)
Batik	5699	4407	22
Avrora	14205	13988	1.5
fop	2634	2218	15
h2	30813	27377	11
jython	37268	36608	1.7
luindex	4190	3458	17.4
lusearch	4040	3496	13
sunflow	6441	5613	12
tomcat	7596	7076	6.8

Table 5.2: Comparing the Performance after the Maxmap changes

Similarly, the size of the buffers used during ROM class building were altered using the same logic. Table 5.3 shows the raw data when the benchmarks were executed on the two versions of the JVM. After the alterations were completed, benchmarks like Batik, luindex and lusearch showed good improvement. In the table, "Improvement (in %)" column shows the improvement that was achieved by changing the ROM class buffers.

Figure 5.2 depicts the performance improvement that was achieved from

Benchmarks	Unchanged JVM (msec)	Maxmap Buffer Imple-mentation (msec)	ROM Class Builder Changes (msec)	Improvement of ROM Class Builder Changes (in %)
Batik	5528.4	4335.5	4591.4	17
Avrora	14296	13072.4	13139.8	8
fop	3006.9	2096.8	2908.8	3.2
h2	29954.6	26328.6	28541.5	4.7
jython	40005.8	31670.3	37005.8	7.5
luindex	4185	3375	3644	13
lusearch	4590.5	3176.5	3769.75	17
pmd	5041.3	4880.85	4932.7	2
sunflow	6426.9	5213.45	6318.7	1.6
tomcat	7400	6984	7105.5	4

Table 5.3: Comparing the Performance after the ROMClass Builder Changes

altering the ROM Class Builder buffers. It can be seen for each benchmark that there was an improvement in performance, but not the same level as the Maxmap buffer. The reason for Maxmap buffer changes having a better performance is that for the altered JVM (from this thesis), the number of reallocations were brought down from 12 to 1 in case of Maxmap buffer whereas for the ROM class buffers it was brought down from 10 to 2 (as seen in Table 5.1). Hence, Maxmap buffer had better performance as during execution the buffer was only resized once which led to a better execution time.

The last set of buffer sizes that were altered are the buffers used during the

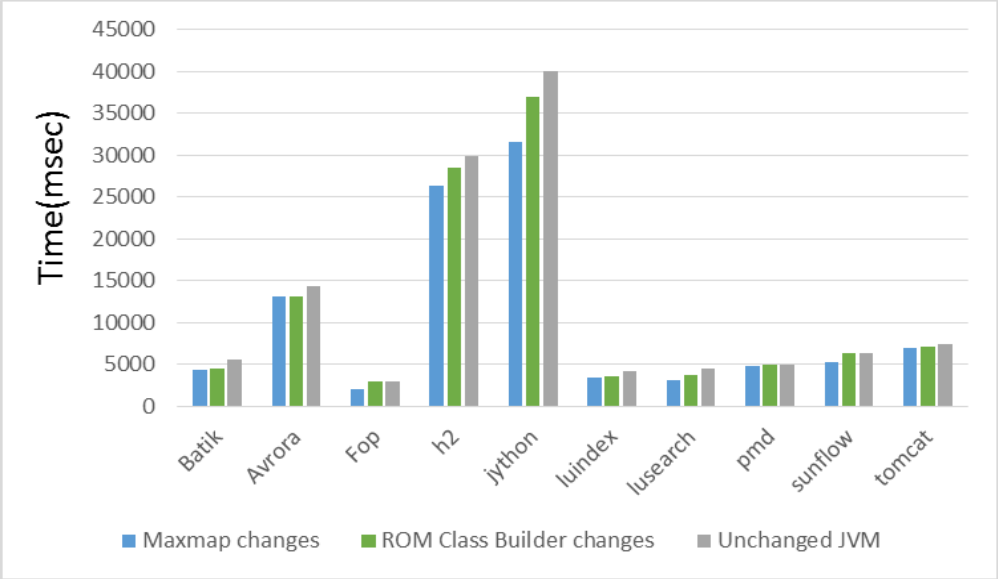

Figure 5.2: Execution Time Comparison Between Altered JVM and Unaltered JVM

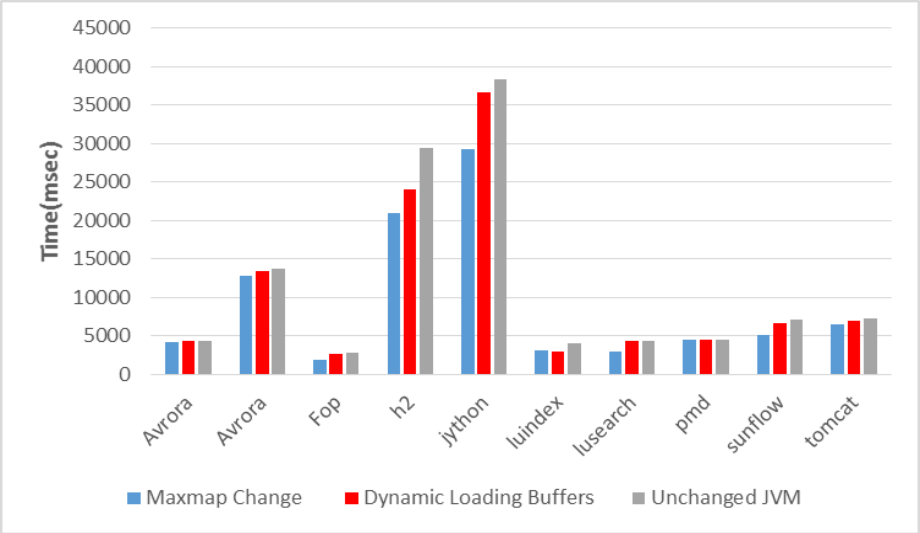

Figure 5.3: Execution Time Comparison Between Altered JVM and Unaltered JVM

dynamic loading of classes. Figure 5.3 shows how the performance of the buffers used during dynamic loading changes with the new implementation. Table 5.4 provides the raw data for Figure 5.3. It can be seen that though there is an improvement when the buffers used during dynamic loading of classes were altered, it is not the same as the improvement from the Maxmap buffer changes because of the fact that the buffers in dynamic loading of classes are reallocated more as seen in Table 5.1. Hence, even after the alterations were made the buffers were being reallocated but not to the same extent as in the unaltered JVM.

The final set of benchmarking that was completed to validate the improvement gained by changing the JVM used the Liberty benchmarks[7]. Both benchmarks were used to validate and Figure 5.4 and 5.5 show the start-up

Benchmarks	Unchanged JVM (msec)	Maxmap Buffer Implementation (msec)	Dynamic Loading Buffer Changes (msec)	Improvement when comparing the dynamic loading buffer changes to the unaltered JVM (in %)
Batik	4397.1	4167.9	4344	1.2
Avrora	13811	12869	13474	2.4
fop	2813	1845	2699	4
h2	29435	21038	23998	18
jython	38334	29322	36627	4.4
luindex	3990	3158	3024	24
lusearch	4364	2962	4294	1.7
pmd	4481	4185	4475	0.2
sunflow	7195	5205	6602	8.2
tomcat	7328	6459	6953	5

Table 5.4: Comparing the Performance after the Dynamic Loading Buffer Changes

Benchmarks	Unchanged JVM (msec)	Maxmap Buffer Imple-mentation (msec)	Dynamic Loading Buffer Changes (msec)	ROM Class Builder Changes (msec)
Tradelite	3421	3338	3220	3206
Daytrader	4162	3920	4061	3927

Table 5.5: Comparing the Performance of each buffer changes using Liberty (in msec)

time recorded using Daytrader and Tradelite respectively. For Daytrader, changes made to the buffers used in dynamic loading led to an improvement of 2.5% whereas changes made to the buffers used in the ROM Class Builder had an improvement of 5.6% and the changes to the Maxmap buffer had an improvement of 5.8%. Similarly for Tradelite, changes made to the buffers used in dynamic loading had an improvement of 2.4% whereas changes made to the buffers used in the ROM Class Builder had an improvement of 5.8% and changes made to the buffers used in the Maxmap buffer had an improvement of 6.2%. Figure 5.4 and 5.5 show the start-up time of each application and it can be seen that reducing the number of reallocations led to a better start-up time.

Table 5.5 shows the raw data for the Liberty benchmarks.

Figure 5.6 shows the performance improvement that was achieved for all the buffers together. The Batik benchmark was used in this case to show the performance comparison. From the figure, it can be seen that the Maxmap buffer (which was reallocated just once) has the maximum improvement as

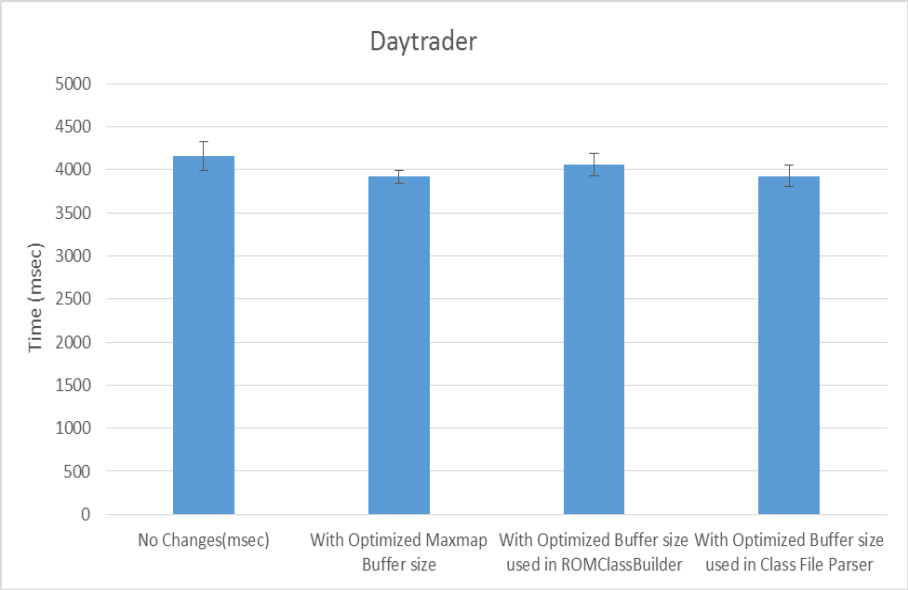

Figure 5.4: Performance Comparison of each Buffer using Daytrader

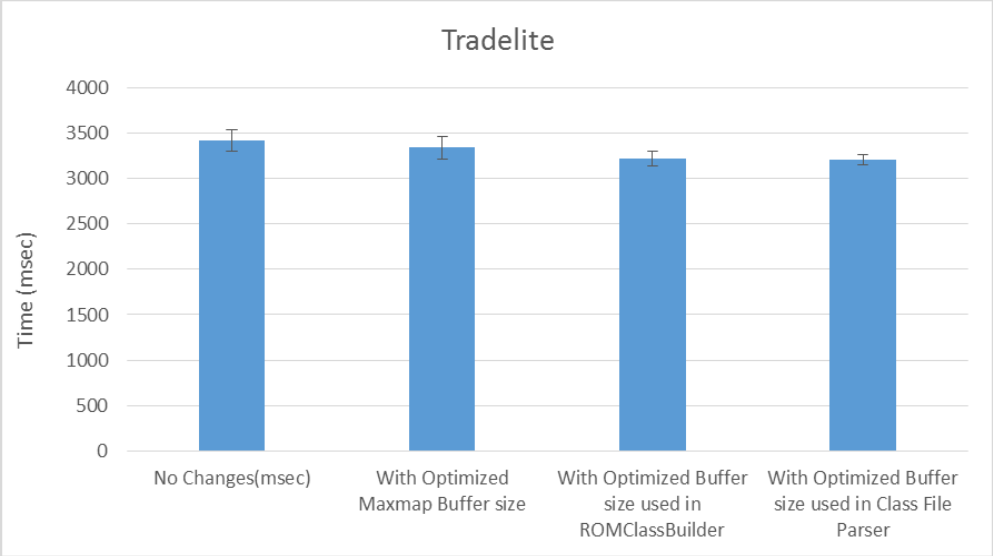

Figure 5.5: Performance Comparison of each Buffer using Tradelite

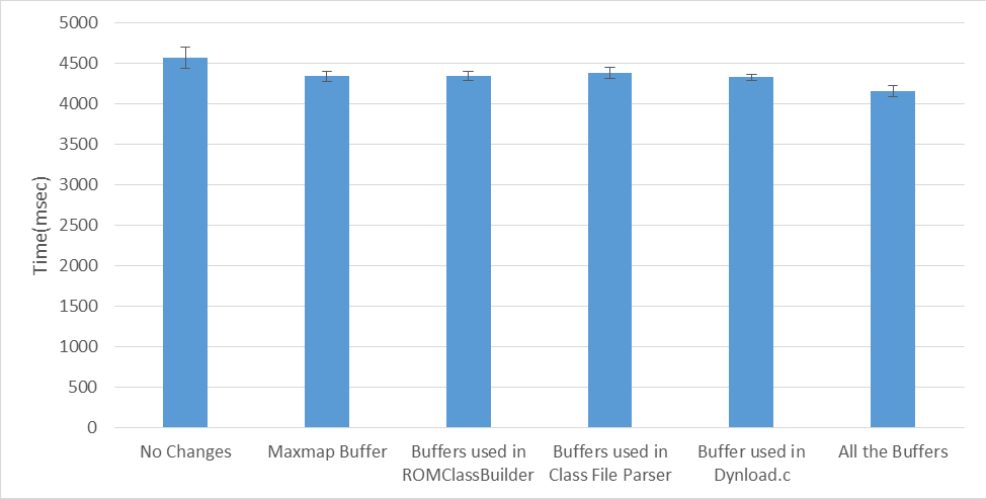

Figure 5.6: Performance after Storing the Buffer Size in the Cache (Batik)

the number of reallocations has substantially reduced when compared to the previous implementation. After evaluating the changes made to each buffer, the execution time for each benchmark improved by a minimum of 3%.

5.2.2 Varying the Heap Size

During this research, the heap size used by an application using OptThruput, GenCon and Balanced GC, was altered in order to maximize performance. This section will provide detailed insight on how changing the GC policies improved the start-up time and memory footprint of different applications. Table 5.6 shows the number of times the heap is resized. It can be seen that, as the heap size is 4 MB initially, there are a number of GCs occurring as well as resizing of the heap. This is time consuming. From the table it can be seen that, if the heap is increased to a larger size then, the number of GCs as well as heap resizing decreases substantially. The heap is increased at the end of the first GC and the reason for the heap expansion is stored. The following are the reasons, why the heap can be increased:

- GC_RATIO_TOO_HIGH: excessive time being spent in GC.
- FREE_SPACE_LESS_MINF: insufficient free space following GC.
- SCAV_RATIO_TOO_HIGH: excessive time being spent scavenging.
- SATISFY_COLLECTOR: continue current collection.
- EXPAND_DESPERATE: satisfy allocation request.

Benchmarks	OptThruput Without using the SCC	OptThruput Using the SCC	GenCon Without the SCC	GenCon Using the SCC
Batik	18	1	12	1
Avrora	5	1	5	1
fop	17	2	45	2
h2	23	10	62	8
jython	63	6	72	6
luindex	7	2	13	3
lusearch	48	14	75	15
pmd	26	10	44	4
sunflow	38	4	49	5
tomcat	20	4	58	6

Table 5.6: Number of Times Heap Resizing Occured

- FORCED_NURSERY_EXPAND: forced nursery expansion.
- HEAP_EXPANSION_FROM_CACHE: heap expansion due to cache.
- default: unknown.

HEAP_EXPANSION_FROM_CACHE was added during this project. The main idea behind changing the heap size was to reduce the number of GCs, so if the heap could be set to the maximum that the application will need, then the number of times the heap is increased can be reduced. In the following section, the two different phases where the heap size is stored in the SCC will be discussed, similarly, the two different phases where the heap is expanded.

In order to store the heap size of an application into the SCC, two phases were

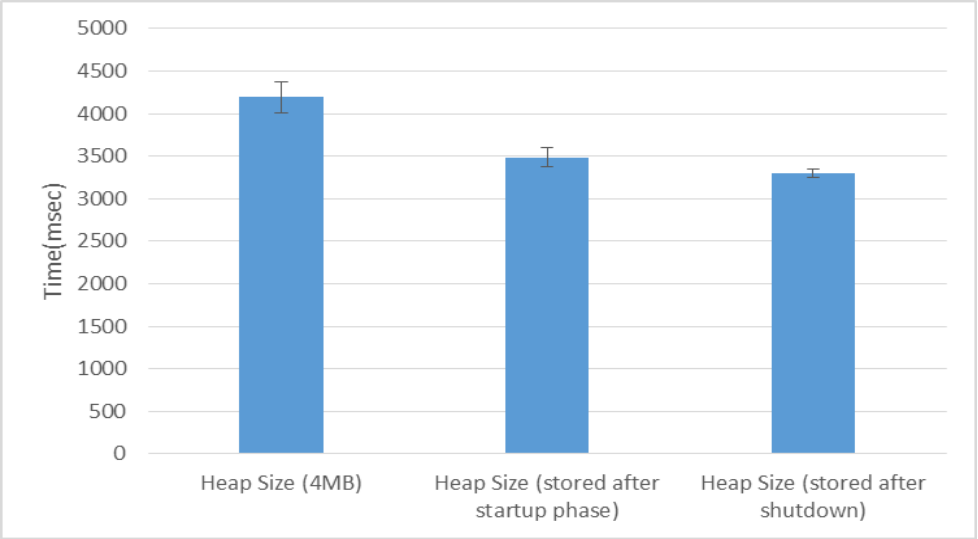

Figure 5.7: Performance Comparison after Storing Heap Sizes from Two Different Phases of an Application (Batik)

chosen: after the end of the start-up phase and during shutdown. Figure 5.7 shows how the execution time of Batik is improved, using the heap size from two different phases. The results show the execution time of an application after the heap size was stored in the SCC. Batik of the DaCapo Suite was chosen because during the lifetime of the application, the heap is expanded drastically causing several GCs to occur. From the figure it can be seen that storing the maximum heap size and reusing it in future can lead to a better execution time. The heap size stored after start-up is almost half of the heap size after shutdown, due to which the application goes through more GCs, leading to a higher execution time.

In Figure 5.7, the results are shown when the GC policy used is GenCon.

When the heap size was stored after start-up, an improvement of 16% was achieved whereas, when the heap size stored during shutdown was used in the next run, a gain of 21% was achieved.

As the heap can only expand when a GC occurs, the application has to wait until the initial heap is full. This was not desirable, as the application had to wait for its first garbage collection to retrieve the desired heap size from the SCC. As soon as the SCC is active an explicit GC is called. The SCC is active right after the heap is initialized hence when the heap does not have any objects allocated yet. The time spent on the explicit GC is negligible as there are no dead objects. Figure 5.8 shows the improvement in start-up time of Daytrader by calling an explicit GC and expanding the heap instead of waiting for a normal GC being called when the heap is full.

Now, the performance of each GC policy change will be shown. Figure 5.9 shows how the performance improved when the OptThruput GC policy was altered. The heap size used during the current run is the maximum heap size that the application used in the previous run. Storing and reusing the maximum heap size from the previous run proves effective as the heap expansion was controlled.

Table 5.7 shows the improvement in start-up for the various benchmarks used. The policy used is OptThruput.

In the case of GenCon, the size of the two spaces (nursery and tenure) is stored in the SCC after the first run. During subsequent runs after the SCC is active, the size of the tenure and nursery space is retrieved and the heap

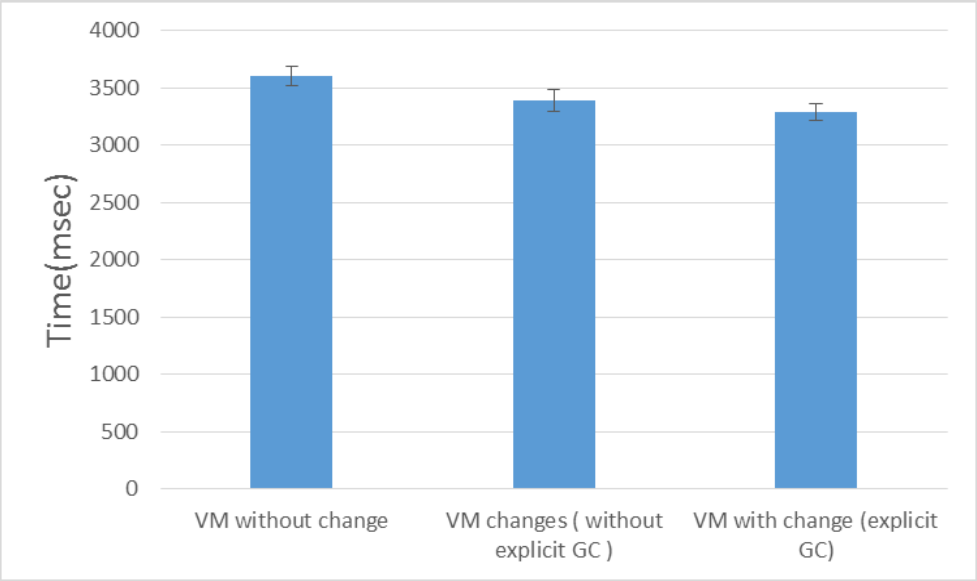

Figure 5.8: Performance Comparison after Calling an Explicit GC right after the SCC is Active (Daytrader)

Benchmarks	Unchanged JVM	JVM with OptThruput changes	improvement in %
Batik	4671.3	4246.9	9
Avrora	13194	12253.25	7
Daytrader	4381	4073	7
Tradelite	3635	3352	8

Table 5.7: Comparing the Execution Time (in msec) after the Optthruput Changes

is expanded. Figure 5.10 shows the improvement in performance when the GenCon policy is altered.

For an application, the memory footprint is directly related to number of

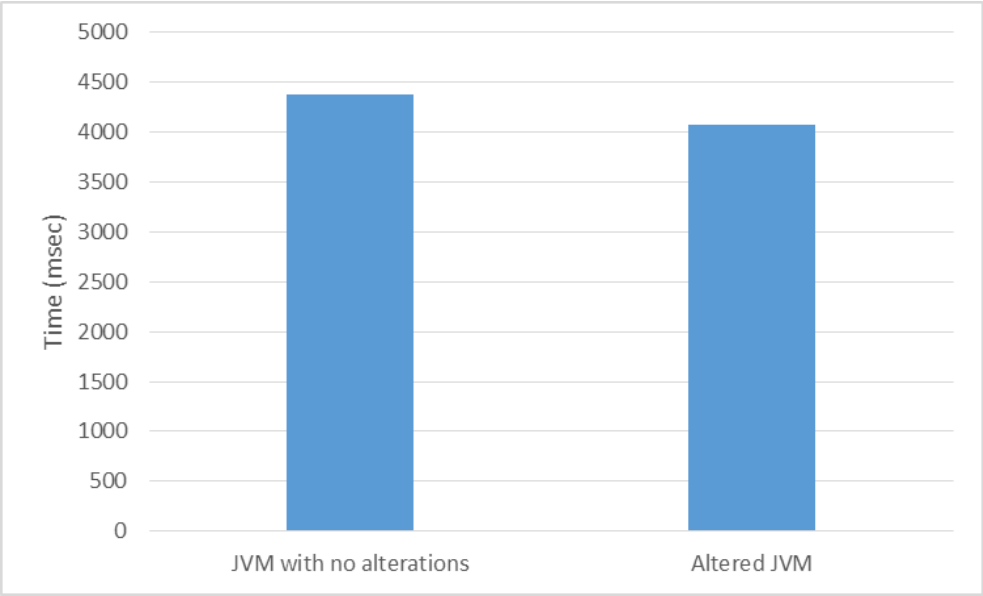

Figure 5.9: Performance Comparison after Changing the OptThruput GC Policy (Tradelite)

objects in the tenure space. During this implementation for GenCon, as the size of the nursery space is increased to the previous runs maximum nursery size, the number of objects that are finally moved into tenure space is diminished, leading to a better memory footprint. Figure 5.11 shows the memory footprint when the policy is GenCon.

The final garbage collection policy that was altered was the Balanced GC. The region size once set, cannot be expanded. So, if the heap has to be increased then it is done by adding new regions. For this GC policy, the total number of regions used by an application was stored instead of storing the heap size. Figure 5.12 shows how changing the Balanced GC policy

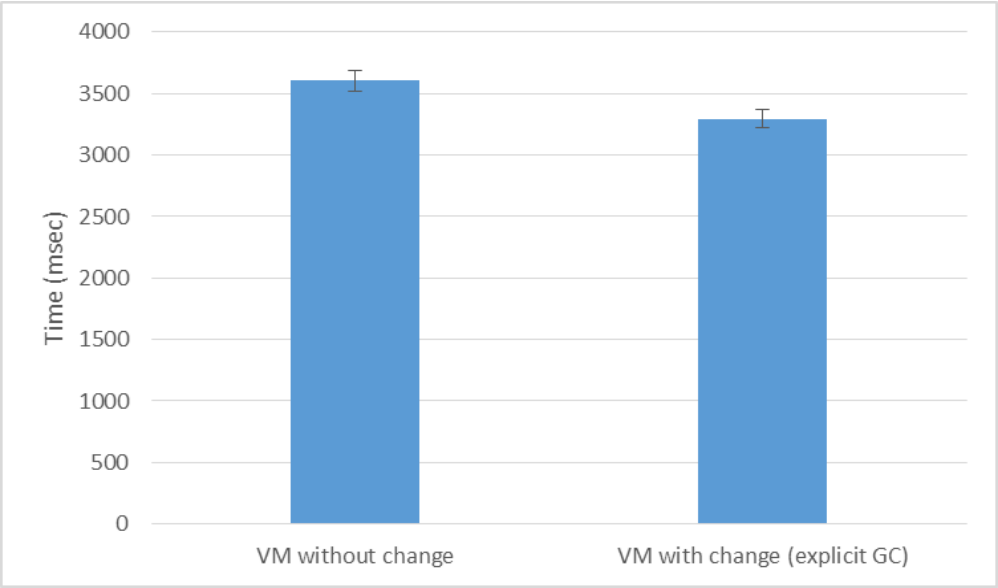

Figure 5.10: Performance Comparison after Changing the GenCon GC Policy (Tradelite)

affected the performance.

According to Neu et al [18], if the GC policy is GenCon and the time taken for garbage collecting is more than 4% of the total time, then the policy should be switched to Balanced. The idea is to see how long each application takes to garbage collect. If it satisfies the criteria stated in [18] then, the appropriate GC policy would be stored in the SCC and during the next run the policy stored in the SCC will determine the GC policy. An issue with this is the SCC is active after the heap structures are already initialized. So, switching the policy on-the-fly will be very difficult. So, for this

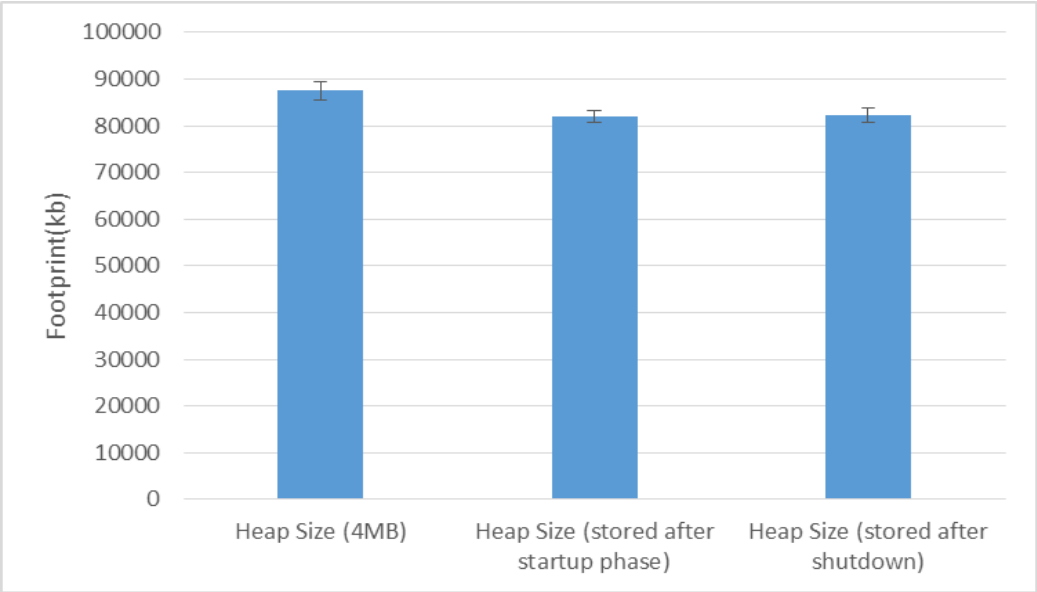

Figure 5.11: Memory Footprint Comparison after Changing the GenCon GC policy (Tradelite)

project instead of switching policies on the fly, the GC time was recorded and if the criteria was fulfilled, then in the next run GC policy was changed accordingly, not automatically but manually.

Table 5.8 shows that when applications had a prolonged garbage collection, switching it to balanced in the next run provided a substantial improvement in performance. The applications were run with GenCon first. From that run, the time spent on the GC policy was collected and if it was over 4% then, the predicted best policy was Balanced. From the table it can be seen that switching the GC policy was fruitful and also, the predictions were correct.

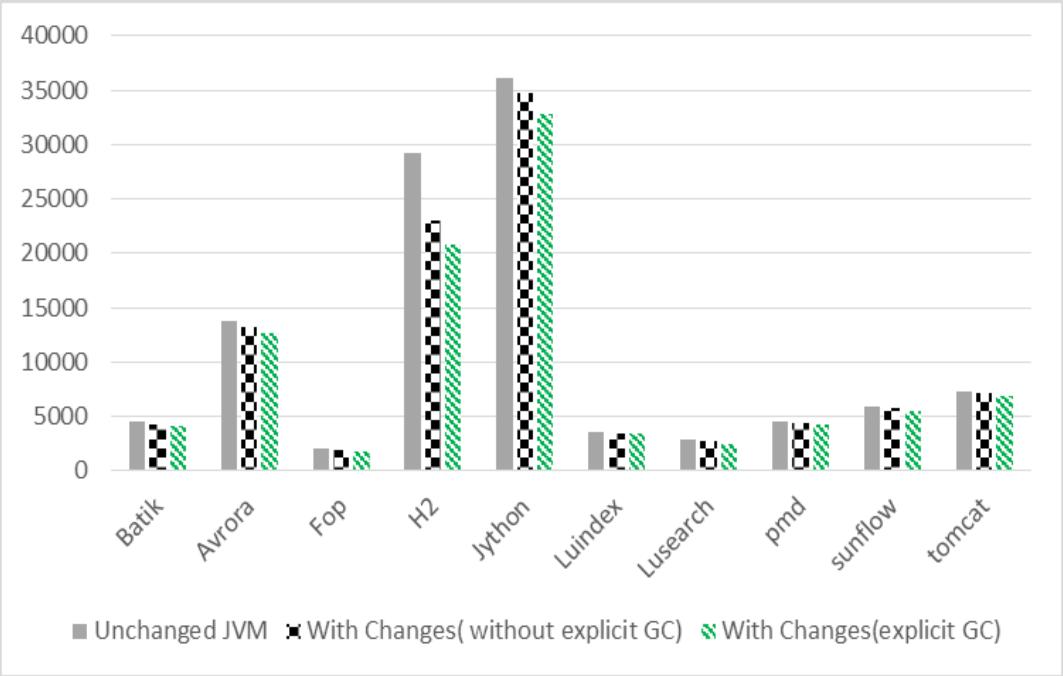

Figure 5.12: Performance Comparison after Changing the Balanced GC Policy (DaCapo Suite)

Every time Balanced GC was predicted, the execution time improved when in the following run, the GC policy was changed to Balanced and the application was re-run. However the problem was that Balanced GC uses an API which gives direct access to the array data on the object heap. If the array is an arraylet (when the size of the array is greater than the size of a region) the data needs to be copied to a contiguous buffer and any changes copied back to the heap. Since an arraylet is only used when the array is bigger than the region size, this is quite a bit of data to copy around and it can make things considerably slower. Solving this problem and making Balanced GC

Benchmarks	GenCon (msec)	Prediction	Balanced (msec)	Time spent on GC(in%)
Batik	4622	Balanced	4244	6
Avrora	13118	GenCon	13122	1
fop	2433	Balanced	1834	12
h2	30283	Balanced	23090	7
jython	33206	GenCon	33983	2.4
luindex	4160	Balanced	3760	4
lusearch	3590	Balanced	2930	12
pmd	4831	Balanced	4551	13
sunflow	7992	Balanced	6822	10.4
tomcat	7256	Balanced	6968	11

Table 5.8: Predicting which Policy to Choose

the default GC policy, is left for future work.

Figure 5.13 shows the throughput when Tradelite was run using the Balanced and GenCon GC policies. It can be seen that the JVM with alterations from this project has overall better throughput.

5.2.3 Pool Size

Class Loaders as well as threads used by the JVM are stored in blocks or pools. During this project, the pool size was altered to save execution time. During the first run, the total number of Class Loaders as well as threads used is stored in the SCC. In the following runs, the pools are initialized based on the total size of these pools, from the previous runs. Figure 5.14 shows how changing the pool sizes impacted performance.

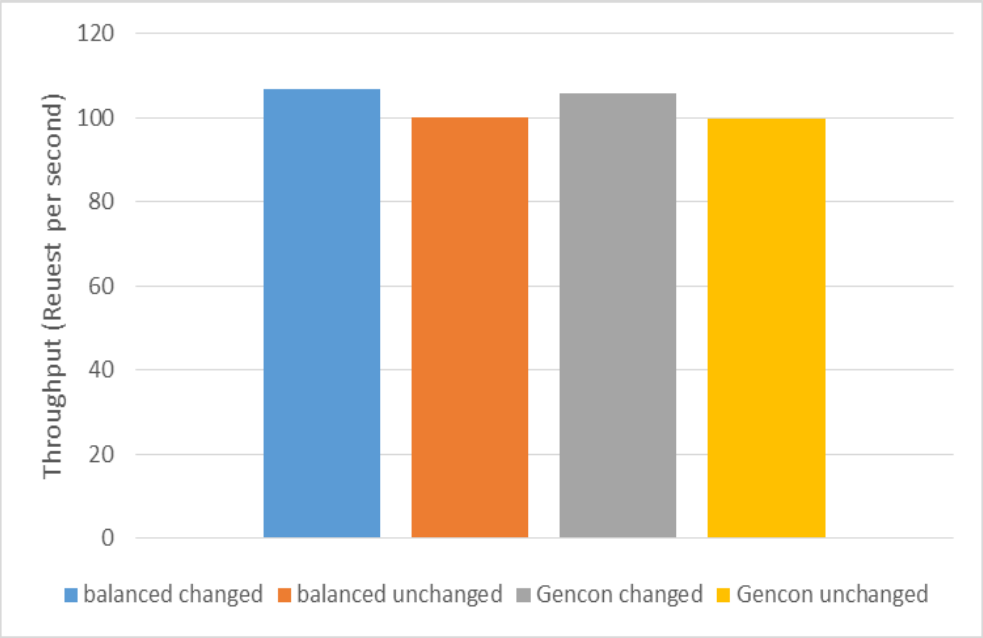

Figure 5.13: Throughput of Tradelite using GenCon and Balanced GC

The first JVM is the unaltered JVM whereas the changed JVM has the pool size changes. Table 5.9 shows the improvement (in %) that was gained.

Finally, Figure 5.15 shows the cumulative effect of all the additions to the SCC. All the buffers, pool sizes and heap size alterations were used for benchmarking performance. The GC policy used was GenCon.

Table 5.10 shows the percentage by which the new implementation improved the performance of the JVM. All applications showed a positive improvement in execution time.

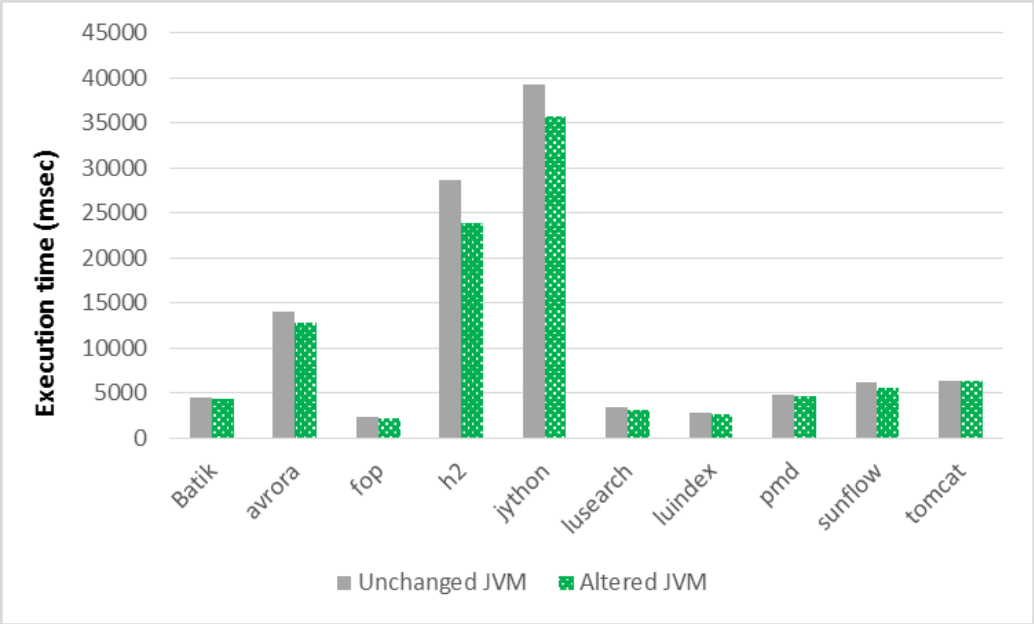

Figure 5.14: Comparing Execution Time of the Two JVMs after Pool Size Changes

Benchmarks	Unchanged JVM	JVM with pool size changes	improvement in %
Batik	4549	4406	3.5
Avrora	13985	12897	7.7
fop	2414	2192	9
h2	28724	23976	16
Jython	39360	35778	9
Lusearch	3507	3211	8
Luindex	2759	2740	0.6
pmd	4805	4685	2.4
Sunflow	6238	5629	9.7
Tomcat	6379	6348	0.4

Table 5.9: Comparing the Execution Time (in msec) after the Pool Size Changes

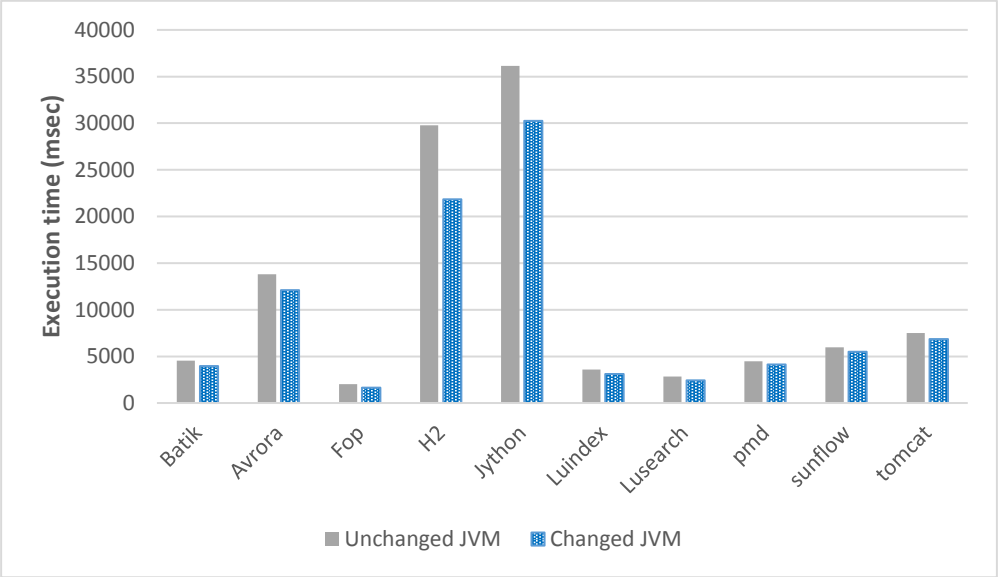

Figure 5.15: Performance Comparison where the Changed JVM has all the Alterations from this Project

Benchmarks	Improvement in %
Batik	14
Avrora	14
fop	30
h2	25
Jython	12
Lusearch	8
Luindex	15
pmd	8
Sunflow	7
Tomcat	6

Table 5.10: Improvement in Execution Time (in %) after all the Changes were Benchmarked Together

Chapter 6

Conclusion and Future Work

The goal for this thesis was to improve start-up time as well as memory footprint of an application, when run on IBM's JVM. This research has shown that if the size of various data structures used during the lifetime of the JVM can be maintained properly, a substantial improvement in start-up time can be achieved. After the data structures are altered, they do not have to start small and grow as needed. Instead, they are set to the maximum size that they grew to in the previous runs. With all the changes made, a minimum of 15% reduction was achieved in execution time whereas, a reduction of 1% was achieved in terms of memory footprint. Figure 6.1 shows the flow of execution for the entire project.While researching this project, the following future sub-projects were identified. As shown in Chapter 5, if the GC policies were altered based on certain criteria then the application may perform better when the GC policy is switched during runtime. Unfortunately, im-

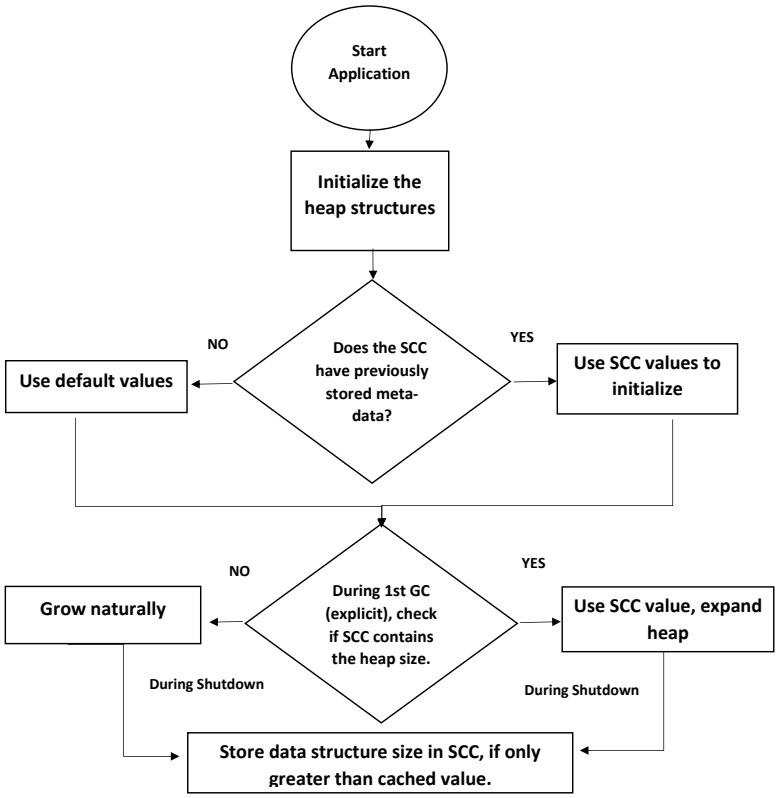

Figure 6.1: Flowchart of the Research

plementing the dynamic switching of GC policies during runtime is a rather large undertaking. Hence, it is an apt project for the future. While looking into changing heap size, it was observed that the heap was being expanded only by the required amount and not by adding a large chunk of memory. Changing the mechanism of adding heap size could also prove to be beneficial. Finally, trying to find more immutable data so that it can be shared among JVMs may be beneficial to reduce the growth of buffers and the overall memory footprint.

Bibliography

[1] D. Bacon, C. Attanasio, V. Rajan, S. Smith, and H. LEE, *A Pure Reference Counting Garbage Collector*, 2007.

[2] Alvin Alexander, *Java stack and heap definitions*, `http://alvinalexander.com/java/java-stack-heap-definitions-memory` [Accessed: 2016-09-08], 2016.

[3] T. Anantham, *Trinath's web applications development - Atom*, `http://trinathswebapps.blogspot.ca/2015/03/jvm-architecture.html` [Online. Last accessed: 2016-Feb-24], 2016.

[4] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al., *The DaCapo benchmarks: Java benchmarking development and analysis*, ACM Sigplan Notices, vol. 41, ACM, 2006, pp. 169–190.

[5] Grzegorz Czajkowski, Laurent Daynès, and Nathaniel Nystrom, *Code sharing among virtual machines*, European Conference on Object-Oriented Programming, Springer, 2002, pp. 155–177.

[6] IBM, *Shared Classes in J2SE 5.0* , 2005.

[7] ———, *About WebSphere Liberty*, `https://www.developer.ibm.com/wasdev/websphere-liberty/` [Accessed: 2016-05-08], 2016.

[8] ———, *Balanced gc policy command line options*, `http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/index.jsp?topic=%2Fcom.ibm.java.lnx.70.doc%2Fdiag%2Fappendixes%2Fcmdline%2Fbalanced_policy_options.html` [Accessed: 2016-09-08], 2016.

[9] ———, *Enhance Performance With Class Sharing*, `http://www.ibm.com/developerworks/java/library/j-sharedclasses` [Accessed: 2016-09-08], 2016.

[10] ———, *Garbage collection in WebSphere Application Server V8, Part 2: Balanced garbage collection as a new option*, `http://www.ibm.com/developerworks/websphere/techjournal/1108_sciampacone/1108_sciampacone.html` [Accessed: 2016-09-08], 2016.

[11] _____, *IBM Knowledge Center - DayTrader*, https://www.ibm.com/support/knowledgecenter/linuxonibm/liaag/wascrypt/l0wscry00_daytrader.htm [Accessed: 2016-09-08], 2016.

[12] _____, *Java Technology, IBM Style: Class Sharing*, http://www.ibm.com/developerworks/java/library/j-ibmjava4/index.html [Accessed: 2016-09-08], 2016.

[13] _____, *Java technology, IBM style: Garbage collection policies*, http://www.ibm.com/developerworks/java/library/j-ibmjava2/ [Accessed: 2016-09-08], 2016.

[14] J. Gosling, B. Joy, G. Steele, G. Bracha and A. Buckley, *The Java Language Specification- Java SE 7 Edition*, Addison-Wesley, 2012.

[15] Kiyokuni Kawachiya, Kazunori Ogata, Daniel Silva, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani, *Cloneable JVM: a new approach to start isolated java applications faster*, Proceedings of the 3rd international conference on Virtual execution environments, ACM, 2007, pp. 1–11.

[16] Nakul Manchanda and Karan Anand, *Non-uniform memory access (numa)*, New York University (2010).

[17] Chuck McManis, *The basics of Java class loaders*, http://www.javaworld.com/article/2077260/learn-java/

`learn-java-the-basics-of-java-class-loaders.html` [Accessed: 2016-09-08].

[18] Nicolas Neu, Kenneth B Kent, Charlie Gracie, and Andre Hinkenjann, *Automatic application performance improvements through VM parameter modification after runtime behavior analysis*, Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014, pp. 147–152.

[19] Bernard Wong, Grzegorz Czajkowski, and Laurent Daynès, *Dynamically loaded classes as shared libraries: An approach to improving virtual machine scalability*, Parallel and Distributed Processing Symposium, 2003. Proceedings. International, IEEE, 2003, pp. 10–pp.

[20] Frank Yellin and Tim Lindholm, *The Java Virtual Machine Specification*, Addison-Wesley (1996).

[21] Benjamin Zorn, *Comparing mark-and sweep and stop-and-copy garbage collection*, Proceedings of the 1990 ACM conference on LISP and functional programming, ACM, 1990, pp. 87–98.

Vita

Candidate's full name: Devarghya Bhattacharya
University attended (with dates and degrees obtained):

West Bengal University of Technology (2008-2012)
Bachelor of Information Technology

University of New Brunswick (2014-2016)
Master of Computer Science

Publications: N/A
Conference Presentations: N/A