Thread-Based, Region-Based Automatic Memory Management

by

Tristan M. Basa

M.S. Comp.Sci., UP Diliman, 2000
B CS, UP Diliman, 1995

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

Masters of Science in Computer Science

In the Graduate Academic Unit of Faculty of Computer Science

Supervisor(s):	Gerhard W. Dueck, Ph.D., Faculty of Computer Science
	Kenneth B. Kent Ph.D., Faculty of Computer Science
Examining Board:	Suprio Ray, Ph.D., Faculty of Computer Science, Chair
	David Bremner, Ph.D., Faculty of Compute Science

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

May, 2017

Abstract

Automatic memory management offers programmers an alternative to unmanaged languages. It unburdens some of the low-level control from human hands. However, this comes at the price of performance. Virtual machines using automatic memory management are prone to long system pause times that degrade the overall performance of the system. Often, the culprit is a mismatch between the nature of the application and the chosen garbage collection algorithm. There are algorithms that already address this issue, such as the *Balanced Garbage Collection* (BGC) by IBM, which uses region-based memory management. Region-based memory management introduces the flexibility of being able to select areas of the heap for collection that will potentially result in the best "return-on-investment."

With the advent of multicore processors, we extend this approach to take advantage of the potential concurrency that can happen among threads during garbage collection. By assigning entire regions to a thread, we are able to select a set of regions that belong to only one thread thereby potentially

allowing other threads to continue execution during garbage collection. We also define thread relationships that will identify other threads that may have to be stopped as well when collection is being done on a given thread.

Dedication

To my Lola Puring...

Acknowledgements

I should like to thank my supervisors Gerhard W. Dueck and Kenneth B. Kent for their continued guidance and support without which I would not be able to finish this thesis.

I should also like to thank Dmitri Pivkine from IBM Ottawa for his important input during our weekly conference calls and prompt emails.

My heartfelt thanks goes to my parents Augusto and Luz Basa who have always been my emotional safety net, not to mention my brothers Ogie, Ramon, Charles, and my sister Ambers who have sustained me in ways no other people can.

Thank you for the faculty and staff especially to Jodi O'Neil of the Faculty of Computer Science of the University of New Brunswick for all the administrative help that they have afforded me.

I should also like to thank Md. Mazder Rahman for not being only a colleague but also a friend.

My gratitude also goes to my other colleagues in the lab for all their constructive criticisms given during our weekly seminars and chance meetings in the hallways.

Last but not the least, thank you to all my Filipino friends here in Canada and in the Philippines who have helped bring Fredericton closer to home.

Table of Contents

Abstract ii

Dedication iv

Acknowledgments v

Table of Contents vii

List of Tables xi

List of Figures xiii

1 Introduction 1

2 Background 4

2.1 Garbage Collection 4

2.2 Garbage Collection Policies 5

2.2.1 Mark-Sweep 6

2.2.2 Reference Counting 7

2.2.3 Generational Garbage Collection 8

2.2.4 Region-based Garbage Collection 11
2.3 Escaping Objects 13
2.4 GC Performance Metrics 15
2.5 *GarCoSim*: A GC Simulator 15

3 Methodology **17**
3.1 Allocation 17
3.2 Collection Set Criteria 19
3.3 Collection 20
3.4 Eden Set Size 20
3.5 Performance Metrics 20
3.5.1 GC Effort 22
3.5.2 GC Return 23

4 Experiments **25**
4.1 Experimental Setup 25
4.2 Input 26
4.2.1 Zombie Objects 28
4.2.2 Suspended Thread Set 29
4.3 Experimental Results 30
4.3.0.1 MyThreads Experiments 45

5 Conclusion and Future Work **51**

Bibliography **59**

A Benchmark Descriptions **60**

A.1 Dacapo 9.12 60

A.1.1 Aurora 60

A.1.2 Batik 60

A.1.3 Fop 61

A.1.4 Lusearch 61

A.1.5 Luindex 61

A.1.6 Pmd 61

A.1.7 Jython 61

A.1.8 Xalan 61

A.2 SPECjvm2008 62

A.2.1 Compiler.compiler 62

A.2.2 Compiler.sunflow 62

A.2.3 Serial 62

A.2.4 Sunflow 63

A.2.5 Xml.transform 63

A.3 MyThreads 65

B Trace File Format **66**

B.1 Allocation 66

B.2 Reference Operation 68

B.3 Class Operation 68

B.4 Store Access 69

B.5 Read Access 71
B.6 Rootset Dump 71

C Source Code **73**

Vita

List of Tables

4.1 Average percentage of related threads. 31

4.2 Workload distribution of threads per benchmark. Work load is defined as the number of operations performed by a thread in the trace file. 32

4.3 Number of garbage collections. The last column is the percentage equivalent of the ratio between the second and third columns. 34

4.4 Number of regions collected. 35

4.5 Number of objects copied. 36

4.6 Bytes copied. 37

4.7 Remset updates. The value *na* denotes not applicable. 38

4.8 Object reference updates. 39

4.9 Collection set remset additions. 40

4.10 Average garbage collection time in seconds. 41

4.11 Simulation times in seconds. 41

4.12 Garbage collection effort. 43

4.13 Bytes freed. 44

4.14 Net regions freed. 45
4.15 Allocation distribution among threads for MyThreads benchmark. 47
4.16 Suspended thread set GC for Table 4.15. 48

List of Figures

2.1 Remembered sets contain pointers that point to objects. . . . 11
2.2 An example of escaping objects: Object 3 escapes to thread B. 14

3.1 Object-splitting. 19

4.1 Number of garbage collections. 34
4.2 Number of regions collected. 35
4.3 Number of objects copied. 36
4.4 Bytes copied. 37
4.5 Remset updates. 38
4.6 Object reference updates. 39
4.7 Collection set remset additions 40
4.8 Average garbage collection time. 42
4.9 Simulation times in seconds. 42
4.10 Garbage collection effort. 44
4.11 Bytes freed. 45
4.12 Net regions freed. 46

4.13 MyThreads is a Java program with five threads performing thousands of allocations concurrently. The x-axis shows the number of operation-steps while the y-axis shows the thread numbers. 49

Chapter 1

Introduction

There has been a renewed interest in virtual machines (VM) since the late 1990s especially in the desktop environment [1]. In a VM that makes use of automated memory management (MM) such as the Java Virtual Machine (JVM), an efficient garbage collection (GC) algorithm is key to good overall performance. One of the main causes of performance degradation in a VM is the long system pause time. A long system pause time is caused mainly by the *stop-the-world* approach to carry out the garbage collection. On a multiprocessor machine, a stop-the-world approach means suspending the execution of all (mutator) threads until the garbage collection is finished [2]. To overcome such a concern, the system pause times can be reduced by minimizing the number of threads to be stopped during the GC. In an ideal scenario, a single thread will be stopped to avoid stopping other threads. Allocation and GC are performed thread-local.

Some approaches make use of *region-based memory management*, which offers a flexible way of managing the memory. The main focus of this research is in the allocation and GC of a specific set of regions belonging to a single thread. When regions are assigned to threads, allocations and collections can be done thread-locally. However, there are thread dependencies that often prevent one thread from being collected independently, such as the presence of *escaping objects*. If an object is referenced by another object from another thread, it is considered to have escaped. Escaping objects can be as high as 42% of all allocated objects (SPECjbb2005)[3]. They are discussed in more detail in the following sections. This study aims to investigate how thread-based, region-based, memory management can potentially minimize the number of threads to stop during collection.

There exist algorithms that make use of region-based memory management, such as IBM's Balanced Garbage collection policy [4]. However, the criteria for the selection of regions in the collection set is based on amortizing the pause times across the number of GCs that will be triggered. The aim of thread-based, region-based, memory management on the other hand, is to minimize the number of threads that would have to be stopped during a GC. As will be discussed later, this does not necessarily translate into shorter simulation time due to increased instances of garbage collections.

Chapter 2 discusses some background literature while Chapter 3 presents

the approach to be taken and details on how thread-based, region-based, memory management can be implemented. Chapter 4 discusses the setup for the experiments as well as the results. Finally, Chapter 5 summarizes the results and gives insights on what could be done next as an extension of this research.

Chapter 2

Background

2.1 Garbage Collection

Prior to managed languages, programmers had to explicitly deallocate unused memory. However, this approach is prone to human errors. By unburdening some of the low-level functionalities from human control, we minimize these kinds of errors. Managed languages give developers several advantages such as increased security, flexibility, fewer lines of code, safer code, and lower cost of deployment. Modern programming languages employ *dynamic memory allocation* wherein objects are allocated on a *heap*. Heap allocation, which can allow programmers to [2]:

- dynamically choose the size of new objects;
- define and use recursive data structures;

- return newly created objects to the parent procedure;
- return a function as a result of another function.

Mutator threads, or application threads, gain access to objects on the heap via *rootsets*. A rootset is a set of references to a *Java Virtual Machine* (JVM) thread's stack and registers during garbage collection time [5].
During a program's run, some objects may no longer be needed and thus lose their reachability from any rootset. These objects are considered garbage and as their number increases, it can lead to out of memory errors. In order to avoid this scenario, garbage collection is needed to reclaim memory occupied by these objects.

2.2 Garbage Collection Policies

There exist several GC algorithms or policies offering flexibility and added capabilities to a managed runtime. Each policy has its own advantages and disadvantages. Depending on the specific goal of the application, a policy is usually chosen such that it will result in the best performance for the system. For example, interactive applications require fast feedback. In this case, a policy with a short collection time is more appropriate. The following subsections present some of the widely used garbage collection policies.

2.2.1 Mark-Sweep

The Mark-Sweep algorithm [6] is one of the four fundamental approaches to Garbage Collection. It is based on the concept of pointer reachability and considered an indirect collection algorithm since it does not detect garbage objects directly. Instead it identifies live objects and assumes everything else is garbage. An example of a direct collection called *reference counting* is discussed in the next subsection.

There are two basic phases in the Mark-Sweep algorithm: the *marking* phase and the *sweeping* phase. The marking phase labels the live objects while the sweeping phase frees up memory space occupied by objects that were left unmarked. The marking phase starts by searching for objects from the *rootsets* of threads. A rootset of a thread contains pointers to objects initially allocated by a thread. By recursively traversing the object pointers contained in the subsequent objects, objects that are reachable from the rootset can be marked and be defined as the set of *live* objects. Unmarked objects are then freed up in the sweeping phase after which, all live objects are unmarked again for the next collection.

Mark-Sweep-Compact is a variation of this algorithm wherein compaction is performed on all live objects after the sweeping phase. Mark-Sweep-Compact is the default GC policy for the IBM's JVM [7].

2.2.2 Reference Counting

Reference counting [8] is an example of a direct collection algorithm since it operates on the objects directly as the number of references to the objects is increased and decreased. Live objects are determined if the number of incoming references to an object is greater than zero. This is tracked by a *reference count*. The reference count is stored as an additional slot in the object's header. Objects whose reference count drops to zero are considered garbage and therefore subject to collection. An issue arises when there are cyclic references, also known as *garbage cycles*. These are objects that reference each other, therefore maintaining a reference count greater than zero. However, none of these objects are reachable from any rootset and therefore should be considered garbage. As the number of cycles increase, the memory footprint increases since objects in the cycles are never freed resulting in less and less available memory.

A simple approach to resolve garbage cycles is to periodically run a tracing collection such as the mark-sweep algorithm discussed in the previous subsection whenever the heap runs low or runs out. Some garbage objects are not part of any garbage cycles and therefore can be caught by reference counting alone. A key observation however is that garbage cycles happen only during reference deletion where the reference count remains greater than zero. This implies that entire object-graph tracing is unnecessary and therefore can be made more efficient. Based on this observation, *partial tracing* can be done on the transitive closure of the subgraph of the object whose reference was

recently deleted but still has reference count greater than zero. An algorithm that takes advantage of this observation and the most widely-adopted algorithm for handling cycles is called *trial deletion*. Attention is only focused on the subgraph that might have created a cycle [9][10][11].

Another approach makes use of an algorithm called *The Recycler* that supports concurrent cyclic reference counting [12][9]. It operates in three phases:

1. Identify objects belonging to a cycle, decrementing their reference count and coloring them grey.

2. Check the reference counts of nodes in these cycles. If an object's reference count is greater than zero, recoloring live grey objects to black, and other grey objects to white.

3. Remaining members of the subgraph that are still colored white are reclaimed.

This policy is useful in reclaiming some memory when some parts of the memory are unavailable, such as in distributed systems [13].

2.2.3 Generational Garbage Collection

Garbage collection is most efficient when there are few live objects in memory. A few observations however, have led to some hypotheses in regards to an object's lifetime [14]. Some observations have become the motivation for another kind of garbage collection policy: the *generational garbage collection*.

One of these observations is that garbage collection is most efficient when there are fewer live objects in memory. The fewer live objects the more memory available for allocation and the more memory available, the fewer collections are required. It is less efficient when long-lived objects are checked by every GC. Another observation that has implications on performance is that most objects tend to die young, which means they should be collected more frequently, and long-lived objects should be collected less frequently.

Generational GC takes its cue from these observations by dividing the heap into *generations* or age groups of the objects. Newly created objects are allocated on the *young* or *nursery* generation. Often, an object's age is measured by the number of collection cycles it has survived. After a certain number of GC cycles, a surviving object is moved to the next older generation. This process is applied for every generation except the oldest.

The youngest generation is collected most often and the older generations are collected less often. The older the generation, the less often it is collected. The ratio of how often collection is done across the generations can be fine-tuned, and so can the size of the each generation. By controlling the size of a generation, we can potentially control the pause times for the collection of that particular generation.

At some point, collection on the young generation will be insufficient to free enough memory as garbage in the older generation starts to accumulate. It is therefore necessary to occasionally perform collection on the older generations as well. Although this approach improves throughput by avoiding repeated

processing of long-lived objects, it comes at a price, that is, it incurs an overhead by keeping track of inter-generational pointers called *remembered sets* (remset). A remembered set is associated per generation. They contain pointers to objects in another generation that have pointers to objects in the current generation. An example is illustrated in Figure 2.1. The *write barrier* code sequence is responsible for determining if a pointer store needs to be remembered or not, and if so, updates the remembered set accordingly [15][16].

Generational collectors should be fine-tuned such that the benefits of using them will outweigh the overhead incurred. There are no strict rules when it comes to fine-tuning generational GC's, so much so that it is sometimes considered a subtle art. Some parameters include the size of the nursery generation, how often should the nursery be collected in relation to older generations, the aging mechanism for objects, and promotion policies for objects.

This policy is preferred whenever the application creates many short-lived objects such as in transaction-based applications wherein objects die after the transaction is completed. Another preferred usage of this policy is whenever an application results in a highly fragmented heap. Moving objects from an often-collected generation to a lesser collected generation tend to minimize fragmentation.

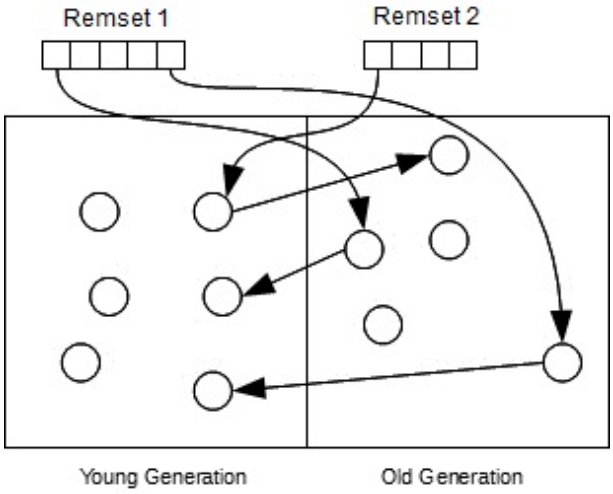

Figure 2.1: Remembered sets contain pointers that point to objects.

2.2.4 Region-based Garbage Collection

The concept of region-based memory management started out as a purely theoretical concept which eventually led to a new memory management technique that is more practical in terms of space and predictability [17].

By splitting the heap into smaller fragments, management becomes more flexible since each fragment can be dealt with under a different policy or mechanism [18].

Region-based memory management is a type of memory management wherein allocated objects are assigned to a region, sometimes called arena, zone, area, or memory context. The regions can be thread-based or longevity-based depending on the purpose. They facilitate allocation and de-allocation with low overhead since deallocation involves merely bumping pointers [19]. Each region marks the area of memory within the region that points to the starting

address of its free space. The size of the free space can be calculated from this marker up to the end address of the region. Allocation is done by moving this free space pointer up according to the size of the object being allocated. If the object size is greater than the regions size, objects can be broken down into smaller chunks to be distributed across other regions. If the whole region is freed, such as in the case of IBM's Balanced GC, deallocation is a matter of moving the free space pointer back to the starting address of the region, and returning the region to the free list.

Region-based MM has been used to improve memory usage in programming languages such as C [20]. By providing programmers low-level control in C, it has become prone to safety violations such as buffer overruns, dangling-pointer dereferences, and space leaks. These can be avoided using automatic memory management.

With region-based MM, automatic memory management can be implemented in such a way that we can choose areas of the heap that will result in a more "balanced" collection time. An example of a region-based memory management system that explores this type of collection policy is IBM's *Balanced Garbage Collection* (BGC) [4]. For better performance, a certain number of regions are targeted at the start of the virtual machine. Depending on the implementation, some systems can also impose a minimum region size. The target number of regions is between 1,024 and 2,048 [4]. Some regions, called the *eden* set, are assigned for allocations. In Balanced GC, the recommended size of the eden set is one-fourth of the current heap size. Collection

is done on a set of regions called the *collection set*. A collection set is the set of regions that meet some criteria for collection. These criteria include the following:

- Eden Regions – all regions in the eden space
- Fragmentaion – highly fragmented regions
- Mortality Rate – objects in the region have a high mortality rate

A *global marking phase* (GMP) runs in the background between collections to help improve performance of a partial GC. During tight memory conditions, a global collection is invoked [4]. In the event system performance degrades, a global garbage collection is necessary such that essentially, a *stop-the-world* GC is in effect.

This type of garbage collection policy is suggested for systems with large memory (at least 4GB). It is also preferred for systems where balanced response-time is important.

2.3 Escaping Objects

The definition of *escaping* objects depends on its context. According to Choi et. al. [21], an object's escape state can be one of the following:

- *GlobalEscape* – An object is shared between methods and threads. For example, an object stored in a static field, stored in a field of an escaped object, or, returned as the result of the current method.

- *ArgEscape* – An object is passed as an argument to a method but does not globally escape during the call.
- *NoEscape* – A scalar replaceable object, meaning its allocation could be removed from generated code.

In this study, we limit the focus on escaping objects shared between threads. This can be identified by finding thread dependencies. Thread dependency based on escaping objects is illustrated in Figure 2.2.

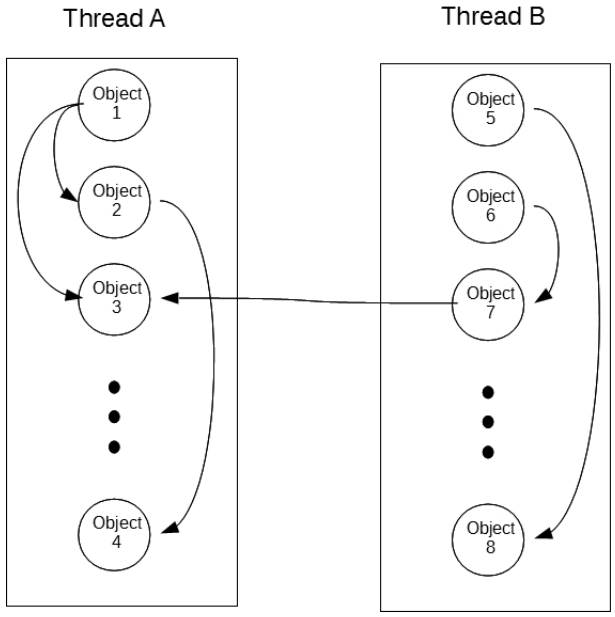

Figure 2.2: An example of escaping objects: Object 3 escapes to thread B.

2.4 GC Performance Metrics

There are several metrics used in evaluating the performance of a garbage collection algorithm [22]. Some of them include:

- **Throughput** – percentage of the summed CPU running time across threads (not including garbage collection).
- **Garbage collection overhead** – percentage of cumulative time spent on garbage collection.
- **Pause time** – time (wall clock) spent while performing a garbage collection.
- **Frequency of collection** – how often does garbage collection occur relative to overall running time of the application.
- **Footprint** – measure of memory consumption such as the heap.
- **Promptness** – time it takes for garbage memory to be reclaimed.

2.5 *GarCoSim*: A GC Simulator

Performance of a garbage collection algorithm depends largely on the requirements of an application. Thus, evaluating its performance entails simulating all the basic memory management operations in a virtual machine [23]. For simulation purposes, an application takes as input a trace file that contains a

series of pre-recorded memory management operations in the JVM. The trace files available have been feature-extracted and can be chosen based on these features. A synthetic trace file generator can also be used to generate trace files based on specific parameters [24]. From this simulation, we can gather statistics which can be used for data analysis and evaluation. Specifically, we can use these statistics to determine how many threads need to suspend during a collection and compare this with the performance of a non-region-based version of collectors such as the traditional mark and sweep.

Implementation of added features was done on top of an existing garbage collector simulator called *GarCoSim* [23]. In terms of allocation, it currently uses the whole heap for allocation or half of it depending on the garbage policy selected. In implementing the thread-based, region-based, GC, the whole heap needs to be divided into fixed, equal-sized regions wherein each region can be assigned to a thread. Modifications were specifically implemented in the *Balanced Garbage Collection* and *RegionBased Allocation* modules.

The proposed thread-based GC builds on top of the above concepts and is further discussed in the following chapter.

Chapter 3

Methodology

Three key areas were identified where the thread-based GC is to be implemented: object allocation, collection set criteria, and collection phase. These key ideas were implemented on top of an existing simulator that uses a non-region-based heap [23]. They are discussed in more detail in the following sections.

3.1 Allocation

Some examples of region-based memory management (such as the IBM's Balanced GC policy), allocations are performed on a set of regions called the *eden set*. The eden set comprises about 25% of the total regions. Different threads can allocate in the same region as long as there is enough space. For a thread-based, region-based allocation, the concept of *thread ownership* is

introduced wherein a thread is restricted to allocate only in regions that it *owns*. Initially, regions are empty and are in a list called the *free list*. The free list is composed of unclaimed regions that are entirely free for allocation. Regions are assigned to the first thread that allocates in the region. Threads that need an extra region for allocation can claim one from this list. Regions are freed or put back in the free list whenever all objects in the region are freed. Algorithmically, a thread follows these steps when allocating an object:

Algorithm 1 Thread-based Allocation

```
if requested space exists in thread's eden set regions then
  allocate object in a thread's eden set region
  return
else if thread's eden set regions < prescribed % of its regions then
  if there are free regions in the free list then
    get a free region
    assign region to the thread
    allocate object in the region
    add region to the eden set
    return
  end if
  return allocation failure
end if
```

Objects that have sizes larger than the region size are split into several objects with each object having a pointer to the next one. Figure 3.1 illustrates how an object that is more than three times the size of a region will be divided into four linked objects.

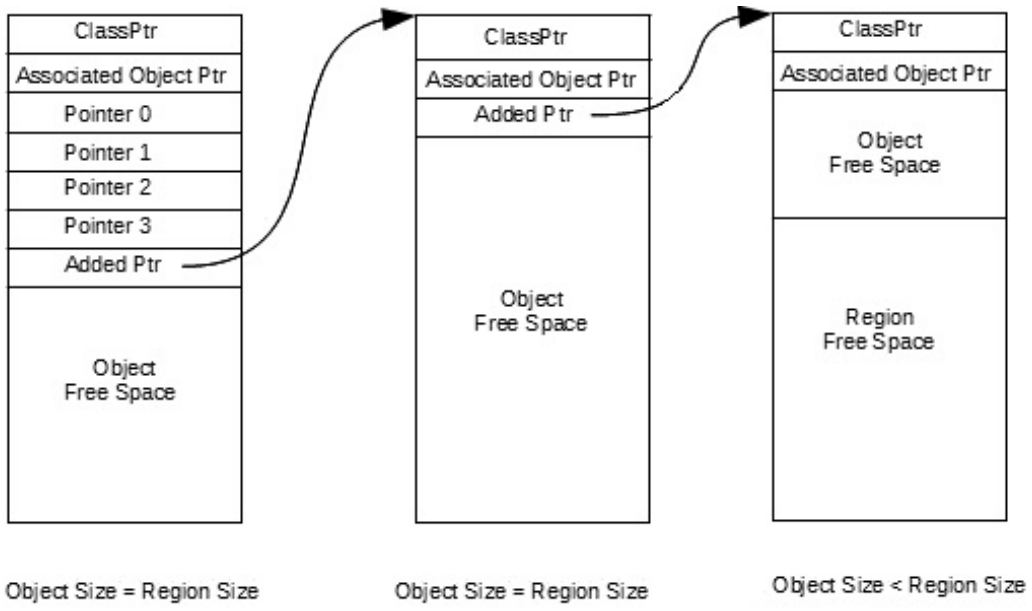

Figure 3.1: Object-splitting.

3.2 Collection Set Criteria

As discussed in Section 2.2.4, the Balanced GC policy imposes criteria per region for selecting which regions to collect. For thread-based, region-based GC, criteria is imposed on the thread for selecting which thread to include in the collection set. In these experiments, the criteria was based on the thread with the highest number of allocations. Regions in the eden set owned by this thread are chosen for inclusion in the collection set.

3.3 Collection

Collection is triggered when there is not enough space in the eden set to accommodate allocation of an object. Upon triggering a GC, we select the thread with the highest allocation count and reset the counter to zero. The thread with the highest allocation count has the highest chance of having the most garbage. This is tracked by keeping an allocation count per thread. The pseudo-code for this approach is given below[1].

3.4 Eden Set Size

In the Balanced GC, the recommended eden set size is 25% of the total regions. In thread-based GC, we arbitrarily set the eden size to 30% of the regions from each thread.

3.5 Performance Metrics

Balanced and thread-based GCs both make use of region-based memory management. For this reason, we define a new set of metrics that focuses more on the specific properties of a region-based memory management system.
Thread-based GC can be viewed as a variant of the Balanced GC. The main difference berween the two approaches lies in the way they allocate objects

[1]In the worst case, we need as many free regions as there are in the collection set. On average, fewer free regions are needed since many objects are usually freed and the remaining live objects copied to the destination regions occupy less space.

Algorithm 2 Collection algorithm

```
T = thread with the highest allocation count
for all region R_e in the eden set do
  if thread owner of region R_e = T then
    add R_e to collection set
  end if
end for
for all live object O_c in the collection set do
  R_d = a non-collection set region of T with enough space for O_c
  if R_d = NULL then
    R_d = get region from the free list
  end if
  copy object O_c to region R_d
end for
return regions in the collection set to the free list
for all objects with pointers to surviving objects in the collection set do
  remove old pointer
  add new pointer to the child's new address
end for
for all remset entries pointing to objects in the collection set do
  if object pointed to was garbage then
    remove remset entry
  else
    remove remset entry
    add object's new address
  end if
end for
allocation count(T) = 0
```

and the way the collection set is chosen. In this regard, some key properties were identified that are relevant to a region-based management system such as remset updates and number of regions freed. Two concepts were introduced for measuring region-based GC performance: the *GC Effort* (GC_e) and the *GC Return* (GC_r)[2].

3.5.1 GC Effort

Garbage collection is a major source of system pause times. For this reason, it is important to measure the overhead involved in performing a GC [25]. Some factors that affect garbage collection are the following:

- **Remset updates** (ru) – these are changes made to the remset data structure associated with each region. Remset update is a two-step process: a deletion and an addition.

- **Number of object reference updates** (or) – measures how many object references need to be updated after objects have been moved to other regions. Similar to remset updates, this also involves a two-step process.

- **Remset addition** (ra) – these are additions to the remsets of the target regions after the copy phase[3].

[2]Pause times are measured in terms of garbage collection times.

[3]We distinguish an update as a two-step process since the number of operations run in the millions and the difference with a single step operation like an addition or deletion can be a factor in the performance.

- **Number of bytes copied** (bc) – measures how much data was copied during the collection. Depending on the implementation, the cost of copying bytes can have a big impact on garbage collection performance.

Given these factors, we formulate a performance measure that takes into account the number of steps performed. For example, updates are weighted twice compared to a simple addition or deletion. The formula then takes the weighted sum of all factors identified above. This measure is defined as GC_{effort} and is given in the equation 3.1. This measure is intended to be hardware agnostic.

$$GC_e = 2(ru + or) + ra + \frac{bc}{8} \tag{3.1}$$

Remset updates (ru) and object reference updates (or) have weights of two since an update is a two-step process: a deletion and an addition. Bytes copied (bc) is divided by eight since in a 64-bit machine, a memory operation is performed eight bytes at a time.

3.5.2 GC Return

Another performance measure that can be taken into consideration is how much resources was freed, or in this case, how much memory was reclaimed. The amount of memory freed will have an impact on the number of collections that can occur. Needless to say, more memory freed results in more memory available for allocations hence, fewer instances of collections. The following

are some statistics that were taken into consideration to measure the GC return:

- **Bytes Freed** – cumulative memory freed in terms of bytes from the objects that were found to be garbage.
- **Regions Freed** – net freed regions. Total number of freed regions minus regions consumed during GCs.

The number of bytes freed per GC can be used to find a lower bound on the number of regions that can be freed per GC, which is equivalent to the floor of the bytes freed divided by the region size. However, due to internal fragmentation, this number can go upwards. In other words, if there are many regions in the collection set that contain only a few objects, the chances of these regions being freed is high, and thus, will result in more regions in the free list.

The following chapter discusses in detail the experiments performed using thread-based GC and the results obtained.

Chapter 4

Experiments

4.1 Experimental Setup

The benchmarks chosen for the experiments came from the DaCapo-9.12 benchmark suites [26] and SPECjvm2008 [27]. Trace files were generated from some of the benchmarks [23]. The benchmarks chosen were those that triggered at least one GC when the simulator was run with a region size of 512KB and a heap size of 512MB. The benchmarks were run in their entirety for both the Balanced GC and thread-based GC. The factors considered in these experiments include:

- Number of garbage collections – garbage collection is a major source of system pauses and ideally should be minimized. However, this metric does not take into account the length of the pauses.
- Size of the collections set – the number of regions to be collected. It

is a coarse-grained measure of the work to be undertaken per GC. The size of the collection set affects all the other metrics considered.

- Number of objects copied – a coarse-grained measure of how much memory was included in the collection set that is not garbage. It is a measure of the efficiency of garbage collection.
- Number of bytes copied – a fine-grained measure of how much memory was included in the collection set but is not garbage. Similar to the number of objects copied, this is also a measure of the efficiency of garbage collection.
- Number of remset updates (deletion/updates) – remset updates measure how many escaping objects were present in the collection set. Escaping objects are not considered garbage and become part of how many objects are copied.
- Number of object reference updates – these are pointers from objects outside the collection set. Similar to remset updates, it is a measure of how much work was done during GC.

It is desirable to have the values for these metrics to be small.

4.2 Input

Memory management operations depend on the memory requirements of different applications. Therefore, measuring performance of an automated

memory management can be achieved with the simulation of the MM operations that happen in an application. A real-time simulation is considered expensive since it requires a copy of the instrumented VM and an actual run of the benchmark applications. An alternative approach is to run the simulation off-line using *trace files*. Trace files contain detailed information on the memory management operations that occurred in the JVM during an application run. They offer the flexibility of being able to develop a tailored simulator that focuses only on the basic MM operations and at the same time, be portable to be compiled and run on different platforms.

The trace files were acquired by instrumenting the Java Virtual Machine (JVM), particularly the memory management module, to capture the basic MM operations [24]. Instrumentation is done by inserting hooks in the JVM code that handle memory-related operations. For each of the operations, a line containing relevant information is written to a trace file which serves as input to the memory management simulator (MMS) [23]. Details of the trace file format is further discussed in Appendix B. A simulator treats the trace file lines as mutator memory operations, allocates space, manages references and performs garbage collections in order to accommodate the requirements of the application. A complete description of the benchmarks is included in Appendix A.

4.2.1 Zombie Objects

Trace files written directly from the instrumented JVM cannot be processed readily due to the presence of *zombie objects*. In principle, once an object dies, it remains dead. There should be no means by which they can be referenced again. However, there are cases found in the trace files where an object has become garbage only to reappear again in the latter part of the the trace. This anomaly has been described as *object resurrection* [28]. For our purposes, we label these objects as *zombies*, not to be confused with zombie objects that refer to actual garbage objects in some literature [29][30][31]. Zombie objects result from pointer stores from C code of the JVM that were not captured by the instrumented JVM since they bypass the Java Native Interface (JNI) [32][33]. Unfortunately, they cause errors during simulations. Attempts to accurately identify zombie objects in the trace files requires the simulation of the object graph, which takes a long time. As a solution, we created another post-processing step that defers rootset deletions until the objects' last access in the trace file. This ensures the object is still reachable at least until its last access. However, this also results in an inaccurate trace file (larger rootsets). This however, does not affect the GC experiments since any GC algorithm will treat an object as live no matter how many pointers it has as long as it is reachable from a rootset.

4.2.2 Suspended Thread Set

From a pessimistic point of view, threads are suspended (during GC) due to the presence or in anticipation of escaping objects. As mentioned, this carries with it a heavy price of long system pause times. An optimistic point of view is the assumption that some of the threads may not contain escaping objects at all and can thus be allowed to continue executing. The thread-based approach offers the possibility of suspending only a subset of the threads. This also offers the possibility of having other threads continue executing during garbage collection. At the very least, the threads that need to be suspended during collection include the triggering thread, the chosen thread for collection (thread with the highest number of allocations, there is a chance that this can also be the triggering thread), and those where escaping objects escape to. Ideally, this subset of suspended threads is small. A Java program (MyThreads.java) was written to determine if we can achieve a small suspended thread set. The program runs five threads concurrently with each thread performing many allocations. This Java program was run under the same instrumented JVM where the DaCapo and SPECjvm benchmarks were run. The trace file produced were also run under the same simulators. Results are shown in Table 4.15 and Figure 4.16 and discussed in Section 4.3.

4.3 Experimental Results

The Balanced Garbage Collection attempts to amortize garbage collection across several collections to mitigate long system pause times. Thread-based GC is a variation of this approach that aims to maximize throughput by allowing possible concurrent execution of threads while collection is being done on a minimal set of threads. Both approaches use a region-based heap but with different allocation and collection set policies.

Similar to Balanced GC, thread-based GC imposes some criteria when selecting regions to be part of the collection set, but instead of the criteria being imposed directly on the regions, we impose the criteria on the threads. The collection set comprises of regions that the chosen thread owns. In case of the Balanced GC, the entire eden set was included in the collection set, while in thread-based GC, only the regions that belong to the thread with the most allocations are included. Both approaches used 30% of the total number of regions as the eden set initially, with thread-based adjusting the size based on the current number of regions that the chosen thread owns.

The benchmark Jython came closest to having similar results between the Balanced GC and the thread-based GC. Further statistics in terms of average percentage of related threads show that Jython has a relatively smaller percentage of related threads during GCs (Table 4.1). The average percentage of related threads is defined as the average ratio between the number of threads where objects in the chosen thread escape to and the total number of threads across all the GCs. Ideally, this should be as small as possible so as to maximize the number of threads that could continue executing. However, should we consider the work load of each thread for the benchmarks, wherein work load is defined as the number of operations each thread performs in the trace file (shown in Table 4.2), we can notice that Jython has only one thread performing most of the work. This renders the thread-based approach similar to the stop-the-world approach. For this reason, the Java program *MyThread* (see Appendix C) was written to investigate the performance of thread-based against Balanced GC given an application that contains mutually exclusive threads.

Table 4.1: Average percentage of related threads.

Benchmark	Average Percentage of Related Thread
batik	47%
pmd	27%
fop	30%
jython	10%
xalan	62%
xml.transform	10%
MyThreads	7%

Table 4.2: Workload distribution of threads per benchmark. Work load is defined as the number of operations performed by a thread in the trace file.

Thread	**batik**	**fop**	**pmd**	**jython**	**xalan**	**xml**
0	3,053	3,053	3,053	3,053	3,052	3,093
1	10,016,963	15,962,350	1,031,418	2,614,761,135	610,032	1,289,646
2	17	17	17	17	14	13
3	2,122	2,569	2,068	2,072	1,998	3,016
4	12	12	12	12	12	12
5	270	189	1,473	4,297	2,564,720	3,560
6	6	512	25,503,221	13	2,615,973	260,818
7	7	11	1,221,996	11	2,476,722	63
8	859,581	724	1,519,728	12	2,601,973	132,776,378
9	58,192	12	1,042,694	945	2,642,073	38,608,229
10	155,199	–	1,379,639	–	2,680,988	7
11	154,498	–	1,310,101	–	2,657,219	185
12	154,498	–	1,141,759	–	2,507,546	14
13	11	–	969,434	–	2,609,437	145
14	115	–	960,730	–	2,542,536	12
15	2,071	–	874,901	–	2,474,234	758
16	12	–	998,219	–	2,596,409	–
17	–	–	1,201,920	–	2,493,976	–
18	–	–	934,490	–	2,623,789	–
19	–	–	3,860,927	–	2,605,117	–
20	–	–	986,357	–	2,647,318	–
21	–	–	4,576,596	–	2,603,444	–
22	–	–	2,006,949	–	2,565,841	–
23	–	–	933,069	–	2,528,349	–
24	–	–	1,548,288	–	2,609,558	–
25	–	–	592,108	–	2,549,249	–
26	–	–	1,500,988	–	2,580,447	–
27	–	–	820,362	–	2,582,033	–
28	–	–	866,103	–	2,627,767	–
29	–	–	1,066,714	–	300	–
30	–	–	90	–	11	–
31	–	–	12	–	13	–
32	–	–	11	–	762	–
33	–	–	2,105	–	–	–

Table 4.3 shows the number of garbage collections that occurred during the entire run of the benchmarks. The last column of the table is the equivalent percentage of the ratio between the Balanced GC and Thread-based GC. Figure 4.1 shows the corresponding graph of this table. It can be noticed that more collections were triggered in the thread-based approach. The thread-based approach allocates objects only in regions owned by the allocating thread. If no region is found, a garbage collection is triggered in the hope that it will free up some regions to be added back to the free list. In other words, a successful allocation entails two conditions being met: there is enough free space in the region, and the region belongs to the allocating thread. In the case of Balanced GC, the first condition only needs to be satisfied. A sharp increase in the number of GCs such as that of Xalan can indicate that the distribution of allocations by the threads among different regions is well-spread especially in the eden set.

These results also show that collecting partially from a thread frees enough memory to allow execution until the end. The benchmark Xalan turned out to have a much higher number of GCs for the thread-based GC. One difference between Balanced GC and thread-based GC is in the size of the collection set. Balanced GC has a more consistent size of the collection set whereas in thread-based GC, it depends on how many regions the chosen thread owns. The fewer regions it owns, the less memory will be freed.

Table 4.4 and Figure 4.2 show the number of regions collected in thread-based GC turned out to be lower than that of the Balanced GC. This can

Table 4.3: Number of garbage collections. The last column is the percentage equivalent of the ratio between the second and third columns.

Benchmark	Balanced GC	Thread-based GC	%
batik	1	1	100%
pmd	4	45	562%
fop	1	1	100%
jython	25	25	100%
xalan	8	153	1,912%
xml.transform	12	13	108%
MyThreads	1	3	300%

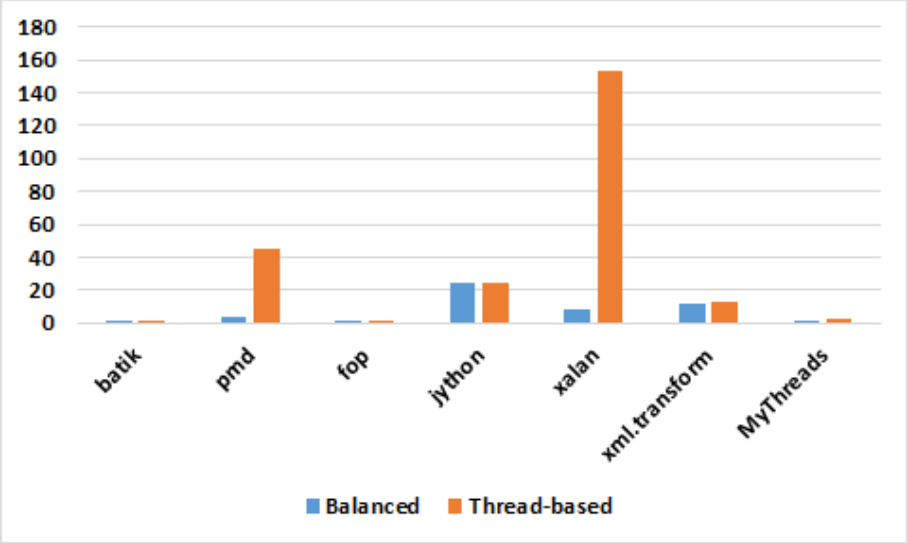

Figure 4.1: Number of garbage collections.

be expected inasmuch as we only include parts of the eden set, that is, only regions in the eden set that belong to the thread chosen for collection will be included in the collection set. In addition, any non-eden region considered to be part of the collection set should also belong to the same chosen thread. This affects the number of bytes copied and remset updates during collection,

which in turn, affects the GC effort. Fewer regions collected results in fewer bytes being copied. Tables 4.6 and 4.7 illustrates this point as well as Figures 4.4 and 4.5.

Table 4.4: Number of regions collected.

Benchmark	Balanced GC	Thread-based GC	%
batik	256	232	91%
pmd	1,110	797	72%
fop	256	252	98%
jython	6,517	6,301	97%
xalan	2,162	2,033	94%
xml.transform	3,320	2,960	89%
MyThreads	512	406	79%

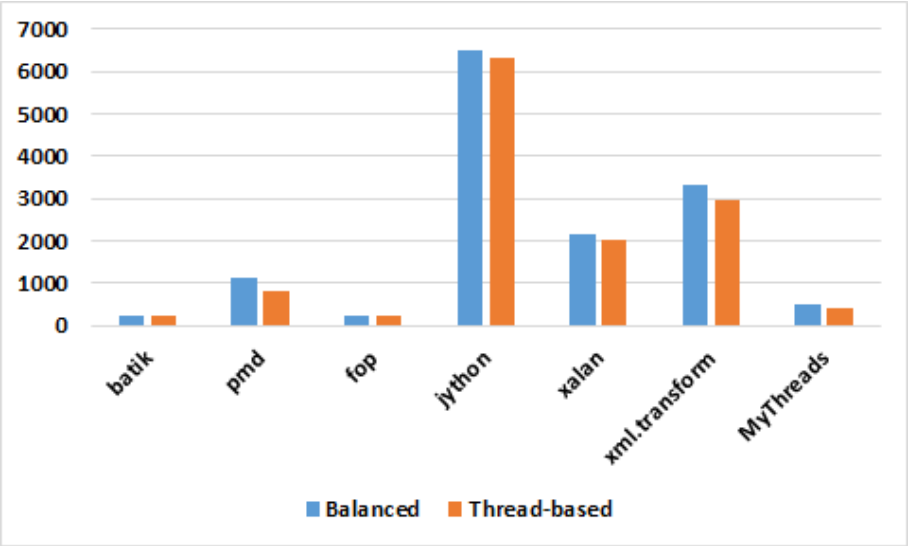

Figure 4.2: Number of regions collected.

Table 4.5 and Figure 4.3 shows results on the number of objects copied. It

can be clearly noticed that there is a lower number of objects copied across all benchmarks. This comes as a result of fewer regions overall included in the collection set.

Table 4.5: Number of objects copied.

Benchmark	Balanced GC	Thread-based GC	%
batik	64,077	54,596	85%
pmd	1,064,595	806,705	76%
fop	187,213	175,794	94%
jython	890,039	561,810	63%
xalan	175,602	120,023	68%
xml.transform	1,767,843	670,808	40%
MyThreads	4,662	238	5%

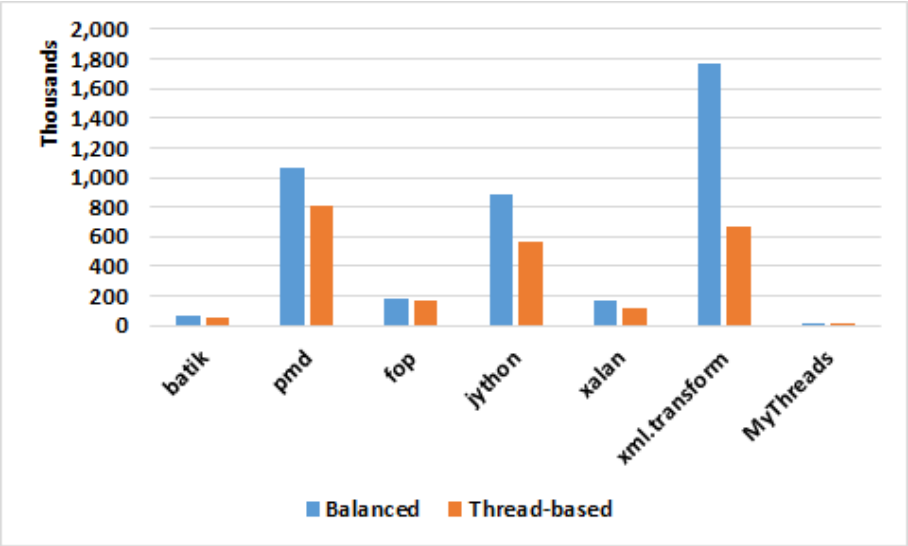

Figure 4.3: Number of objects copied.

Table 4.6 and Figure 4.4 show the experimental results on the total number of bytes copied.

Table 4.6: Bytes copied.

Benchmark	Balanced GC	Thread-based GC	%
batik	14,145,656	13,032,328	92%
pmd	81,309,896	53,556,256	66%
fop	13,029,056	12,262,512	94%
jython	71,192,720	45,915,568	64%
xalan	66,645,416	55,313,096	83%
xml.transform	137,145,504	54,369,640	40%
MyThreads	348,888	7,616	2%

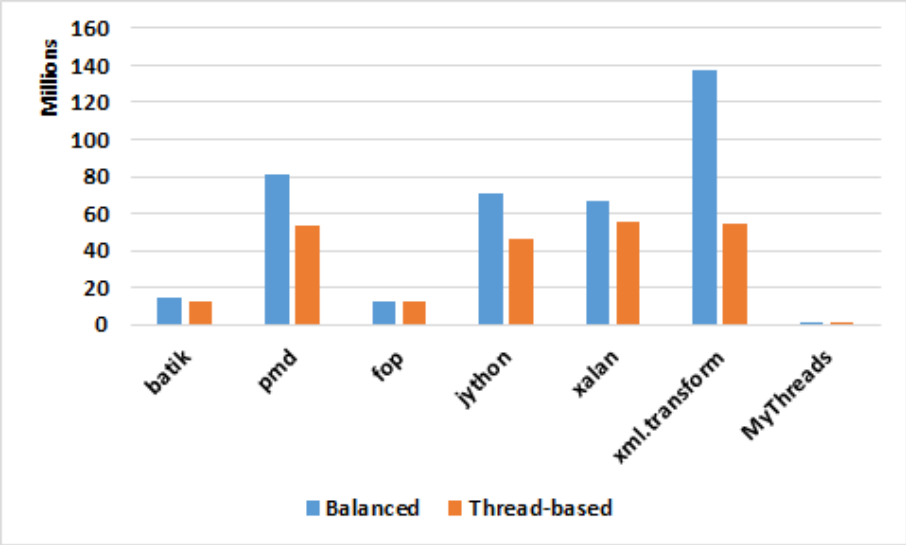

Figure 4.4: Bytes copied.

Table 4.7 shows data on total remset updates for both the Balanced and Thread-based GC. These figures include remset deletions. Since remset updates measure updates on regions outside the collections set, a sharp increase in the benchmarks Pmd and Xalan can indicate that regions chosen for collection by the Balanced GC has fewer escaping objects compared to

thread-based.

Table 4.7: Remset updates. The value *na* denotes not applicable.

Benchmark	Balanced GC	Thread-based GC	%
batik	0	54,872	na
pmd	2,294,166	38,417,816	1,675%
fop	0	4,954	na
jython	35,085,087	37,643,331	107%
xalan	1,352,399	64,262,782	4,752%
xml.transform	3,932,187	9,038,310	230%
MyThreads	0	4,641	na

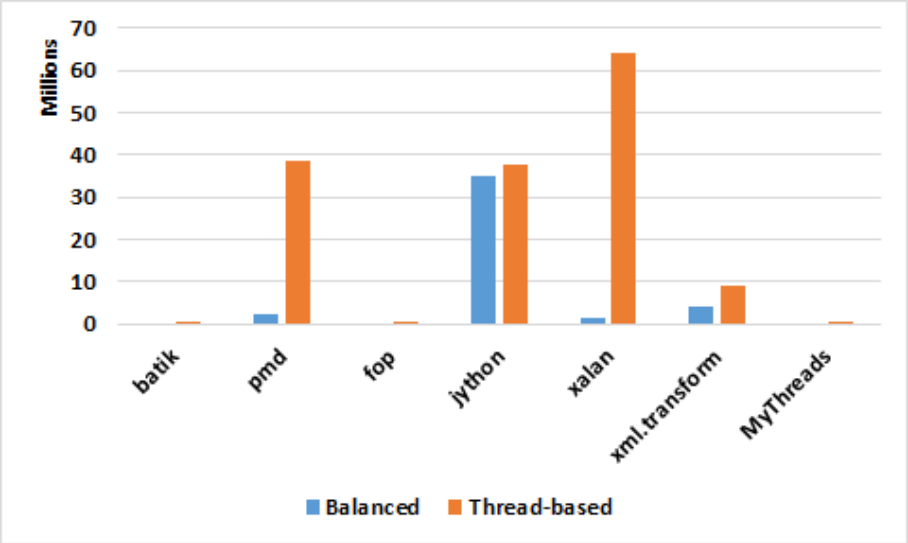

Figure 4.5: Remset updates.

Tables 4.8 and Figure 4.6 show updates on object references. Object reference updates are made on pointers of an object that references live objects in the collection set that were eventually moved to a new region during garbage

collection. The results show a trend towards fewer object reference updates in thread-based GC compared to Balanced GC. This can also be attributed to fewer regions and fewer objects copied.

Table 4.8: Object reference updates.

Benchmark	Balanced	Thread-based	%
batik	1,704,011	1,592,040	93%
fop	3,090,062	3,091,101	100%
pmd	8,752,237	3,105,367	35%
xalan	8,995,994	4,258,987	47%
jython	48,781,786	42,215,406	87%
xml.transform	23,809,992	16,442,680	69%
MyThreads	19,532	3,312	17%

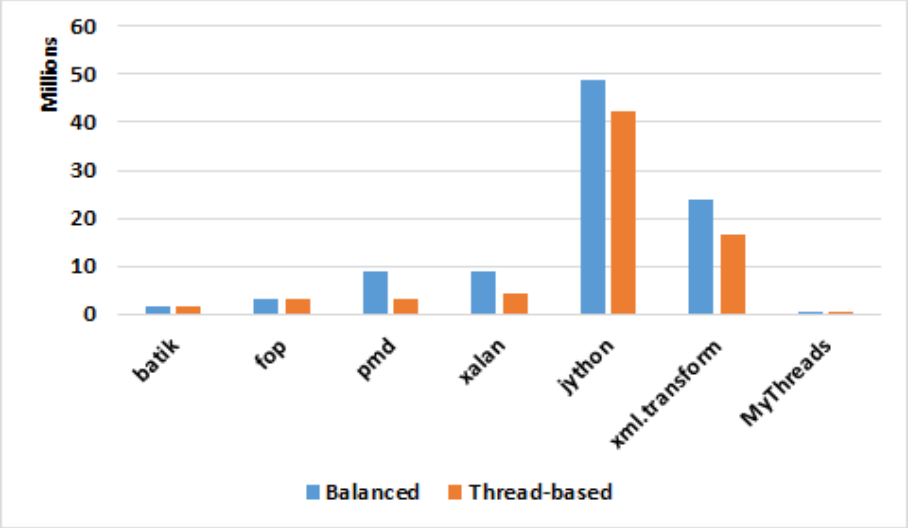

Figure 4.6: Object reference updates.

Table 4.9 and Figure 4.7 shows results on the number of remset additions. We differentiate them from the remset updates in Table 4.7 and Figure 4.5

in that these are entries added to the remsets of newly acquired destination regions of live objects during the copy phase of garbage collection.

Table 4.9: Collection set remset additions.

Benchmark	Balanced	Thread-based	%
batik	120,488	84,722	70%
fop	786,248	786,885	100%
pmd	2,048,759	704,874	34%
xalan	1,182,095	1,241,315	105%
jython	13,619,499	16,236,157	119%
xml.transform	4,969,637	1,509,629	30%
MyThreads	1,533	1,656	108%

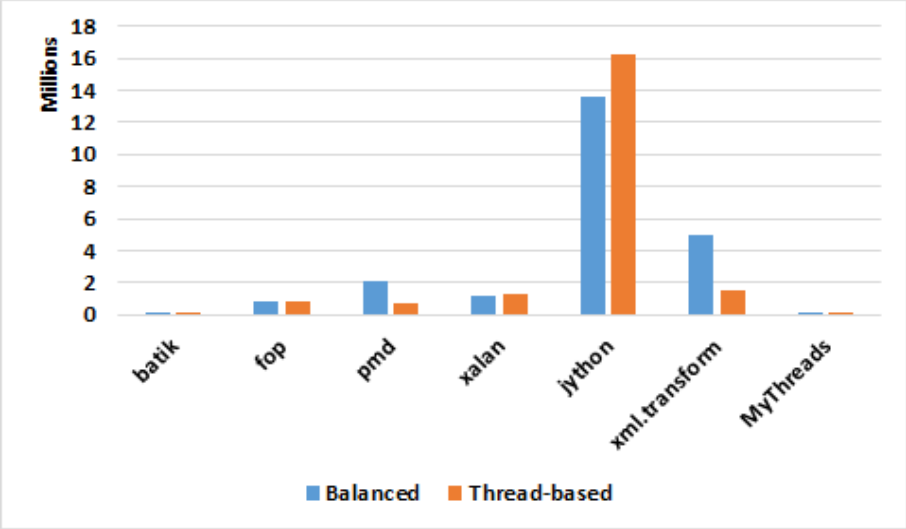

Figure 4.7: Collection set remset additions .

Table 4.10 and Figure 4.8 show the average garbage collection time in seconds while Table 4.11 and Figure 4.9 show results on the overall simulation time in seconds (wall clock). Balanced GC tends to have fewer GCs, longer

collection times per GC, and a shorter overall simulation time while the thread-based GC tends to have more GCs, a shorter collection time per GC, and longer overall simulation time. Simulation time is highly dependent on the implementation of the simulator and is intended to be a rough estimate. If Balanced GC amortizes collection across several instances to achieve balanced collection times, the thread-based approach pushes this notion further by having more GCs. Although the cumulative collection time is generally longer, the average collection time per GC is shorter. The cumulative time takes into account the cumulative overhead involved with every GC.

Table 4.10: Average garbage collection time in seconds.

Benchmark	Balanced GC	Thread-based GC	%
batik	11.97	8.96	75%
pmd	17.47	6.58	38%
fop	14.09	10.36	74%
jython	18.16	14.76	81%
xalan	11.97	5.03	42%
xml.transform	15.66	11.71	75%
MyThreads	16.69	7.08	42%

Table 4.11: Simulation times in seconds.

Benchmark	Balanced GC	Thread-based GC	%
batik	52.72	49.82	94%
pmd	344.87	590.74	171%
fop	73.79	72.33	98%
jython	1,766.8	1,771.39	100%
xalan	389.8	1,119.66	287%
xml.transform	893.5	906.76	101%
MyThreads	128	158.51	124%

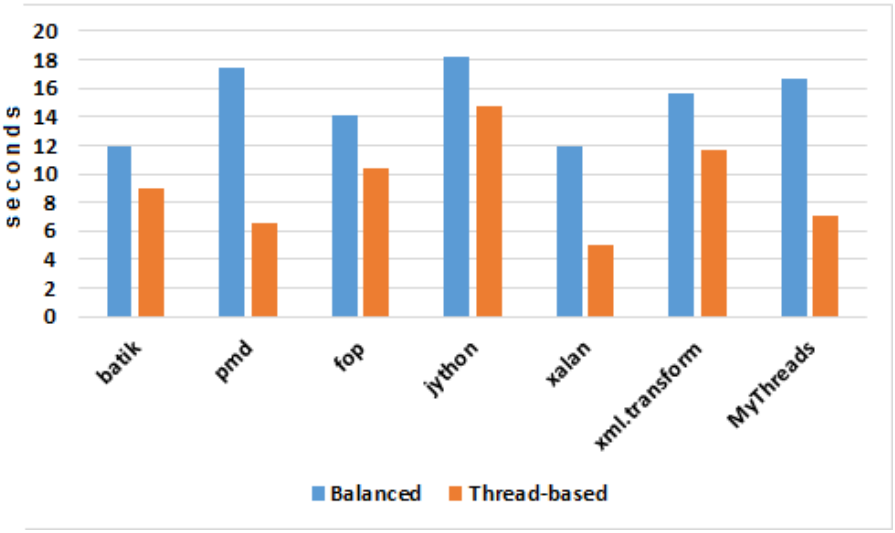

Figure 4.8: Average garbage collection time.

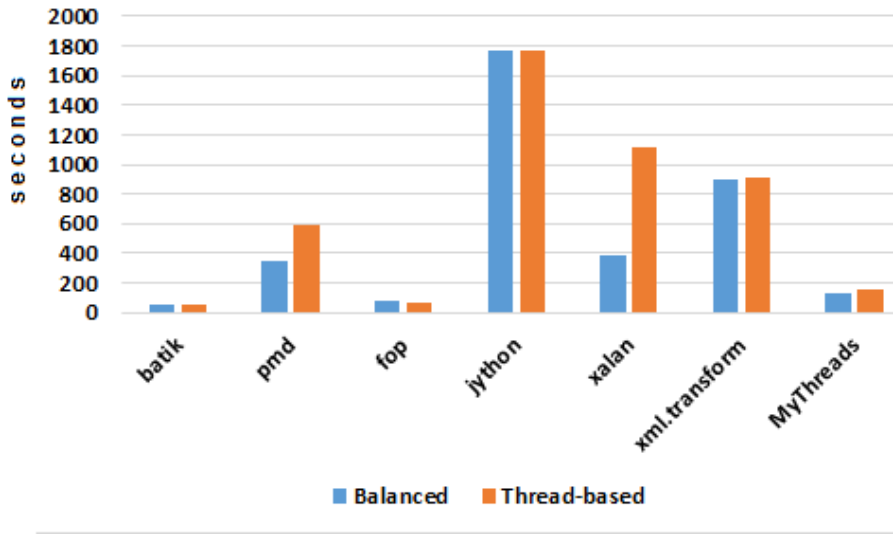

Figure 4.9: Simulation times in seconds.

As performance metrics, we measure the amount of "effort" a garbage collection has incurred over the entire run. The amount of effort is based on statistics gathered on key attributes presented in the experimental results

section. Table 4.12 and Figure 4.10 show results on the calculated GC effort based on the formula defined in Section 3.5. Although the results show more effort involved in the thread-based approach compared to Balanced GC (there are zero values for batik and fop for Balanced GC since there were no collections triggered), other factors such as the number of bytes copied, number of regions in the collection set, and number of objects copied are less than in the thread-based.

It can also be noticed that the best GC effort improvement happened in the benchmark MyThreads. This can be attributed to the lack of escaping objects between the threads, which is a big factor based on the formula. The large discrepancies between Balanced GC and thread-based as shown in the benchmarks Pmd and Xalan are due to the large discrepancies in remset updates.

Table 4.12: Garbage collection effort.

Benchmark	Balanced GC	Thread-based GC	%
batik	5,296,717	5,007,637	95%
pmd	34,305,302	90,445,772	264%
fop	8,595,004	8,511,809	99%
jython	190,252,335	181,693,077	96%
xalan	30,209,558	145,198,990	481%
xml.transform	34,739,183	59,267,814	171%
MyThreads	84,208	18,514	22%

In terms of GC Return, Table 4.13 and Figure 4.11 show results on the number of bytes freed and Table 4.14 and Figure 4.12 show results on the total net regions freed. For both the number of bytes freed and the number

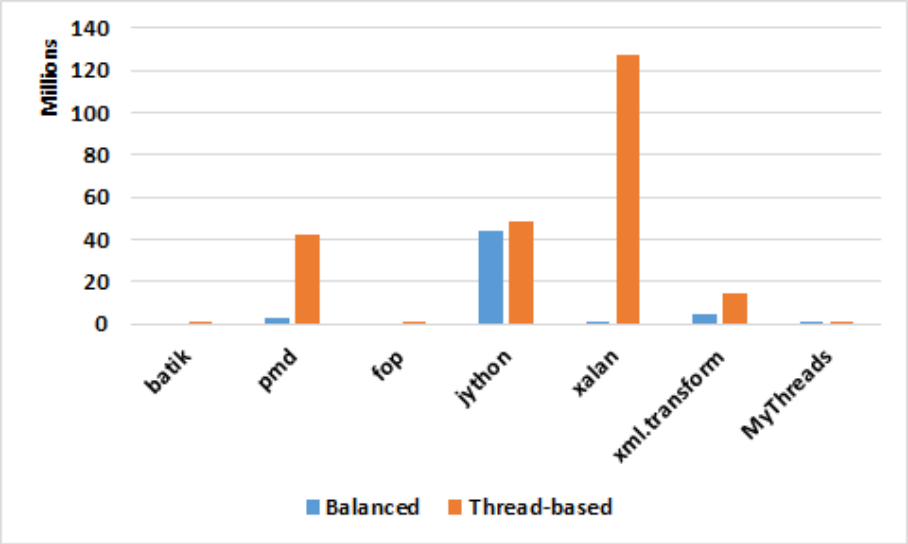

Figure 4.10: Garbage collection effort.

of net regions freed, the values are slightly less in thread-based compared to the Balanced GC. This could be attributed to fewer regions being included in the collection set which generally results in fewer objects and fewer bytes scanned during each GC.

Table 4.13: Bytes freed.

Benchmark	Balanced GC	Thread-based GC	%
batik	120,070,232	107,964,760	90%
pmd	498,944,432	356,464,496	71%
fop	121,187,136	119,856,784	99%
jython	3,324,403,088	3,257,576,912	98%
xalan	1,063,246,224	971,418,952	91%
xml.transform	1,589,541,248	1,497,507,280	94%
MyThreads	133,868,824	106,235,552	79%

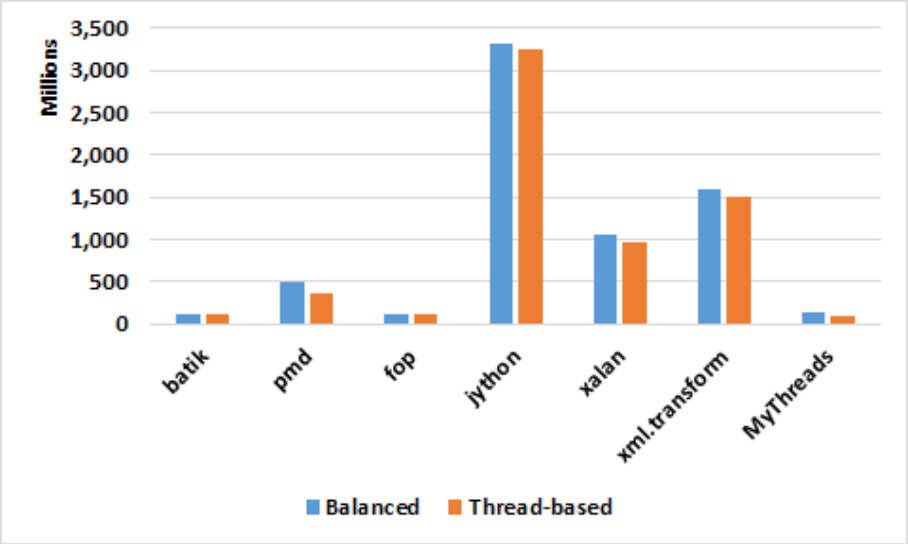

Figure 4.11: Bytes freed.

Table 4.14: Net regions freed.

Benchmark	Balanced GC	Thread-based GC	%
batik	212	186	88%
pmd	670	551	82%
fop	230	226	98%
jython	5,747	5,642	98%
xalan	2,015	1,755	87%
xml.transform	3,034	2,992	99%
MyThreads	255	266	104%

4.3.0.1 MyThreads Experiments

A proposed advantage of the thread-based approach to garbage collection is the potential concurrency that can happen during a GC. By having other threads continue to execute while a GC is performed on another thread, long system pause times caused by the stop-the-world approach can be mitigated.

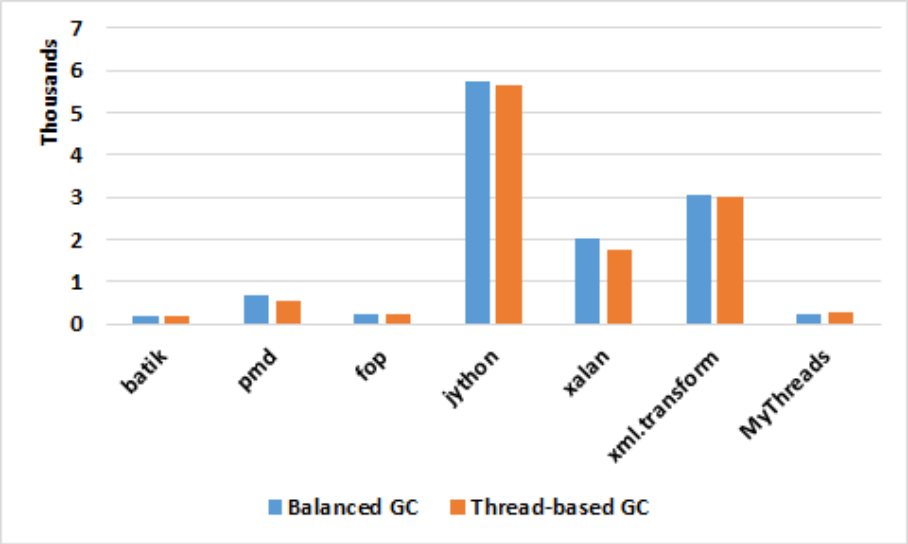

Figure 4.12: Net regions freed.

One issue that prevents this from happening is the presence of escaping objects. Escaping objects define the relationships between threads from the point of view of garbage collection. If a thread is chosen for collection, threads where its objects "escape to" need to be suspended as well. If escaping objects escape to many different threads, then many threads would have to be suspended during collection, and in the worst case, can have the same effect as the stop-the-world approach.

This is the main motivation behind the following experiments. A Java program was written (MyThreads.java) to purposely investigate if a scenario can happen wherein there are few, if any, escaping objects between threads. Five threads were instantiated using different classes. Each thread was tasked to perform object allocations on a local variable a number of times frequently

enough to trigger a GC. The resulting trace file allocation distribution is presented in Table 4.15. Thread T_4 was tasked to perform many allocations which resulted in about four million object allocations in the JVM. As expected, the simulator chose T_4 as the thread to be collected when a GC was triggered. During collection, remsets of regions in the collection set were inspected to determine if there were objects escaping to other threads, which would imply suspending those other threads as well. The worst-case scenario is having objects escape to all the other threads, which would have the same effect as a stop-the-world approach. As expected, the simulator found no escaping objects during the collection. This supports the hypothesis that in a thread-based GC, other threads can continue execution while the GC is being performed on a minimal number of threads.

Table 4.15: Allocation distribution among threads for MyThreads benchmark.

Thread No.	**Number of Allocations**
0	620
1	8,363
2	1
3	434
4	2,500,000
5	1,500,000
6	750,000
7	2,000,000
8	250,000
9	31
10	2
11	115

Table 4.16: Suspended thread set GC for Table 4.15.

GC No.	Triggering Thread	Chosen Thread	Related Threads
1	4	5	1
2	7	4	1
3	4	4	1

Whenever there are escaping objects, there are implied synchronization issues. The experiment shown in Figure 4.13 is an analysis of how threads from the MyThreads benchmark can theoretically continue their independent executions in the absence of a GC and escaping objects. Threads four to eight, as mentioned, are specifically written to allocate locally. The x-axis corresponds to the line steps (lines read from a file) while the y-axis corresponds to the thread number. A line represents a sequence of operations from a thread's trace file. A continuous line means the operations can proceed without waiting for objects from another thread (escaping objects). By adding GC times on the steps of the threads chosen for GCs in Figure 4.13, the thread that was collected most frequently and with the highest final step (as measured by the x-axis) can serve as a rough estimate of the makespan. The advantage of thread-based GC becomes more apparent when GCs occur more often on threads that have fewer steps to finish and less often on threads that take more steps to finish. This will have the effect of balancing out the running times for each thread, thus minimizing makespan. If the same number of collections occur (with the same GC times) using the Balanced GC, all GC times will be added to the running times of all

of the threads (since Balanced GC employs a stop-the-world approach) and effectively, to the makespan as well. In other words, the makespan of Balanced GC serves as an upper bound to the makespan of thread-based GC. In addition, thread-based GC offers the flexibility of proactively choosing idle threads for collection.

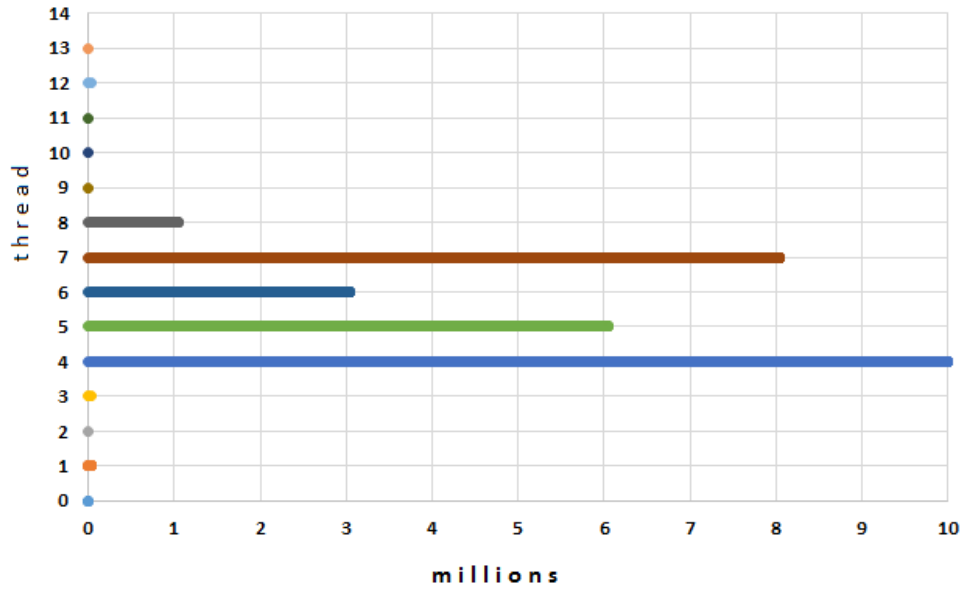

Figure 4.13: MyThreads is a Java program with five threads performing thousands of allocations concurrently. The x-axis shows the number of operation-steps while the y-axis shows the thread numbers.

In summary, restricting object allocation to regions owned by the allocating thread resulted in increased occurrences of garbage collection in thread-based GC as compared to Balanced GC. This in turn, resulted in fewer regions included in the collection set, which resulted in lower values for thread-based GC across the different experiments. In terms of GC effort, thread-based

GC results were higher due to increased remset updates that had to be performed on non-collection set regions (smaller collection sets mean more non-collection set regions).

MyThreads experiments show there can be few escaping objects between threads. This suggests that in terms of escaping objects, GC can be performed on a thread without hindering other threads from continued execution.

Chapter 5

Conclusion and Future Work

In automatic memory management, long system pause time is an undesirable characteristic. A common culprit is the stop-the-world approach employed by some garbage collection algorithms. The stop-the-world approach entails suspending the execution of all threads thereby rendering the system in a pause state. One obvious solution to this problem is to allow some of the threads to continue execution while performing collection on a subset of threads. However, there are certain cases that may prevent this scenario from happening such as the presence of escaping objects.

In this study, we investigated the feasibility of a thread-based approach to garbage collection. The goal is to mitigate the long system pause times caused by the stop-the-world approach. We have identified and developed some metrics to measure the performance of the thread-based garbage collection in comparison to the Balanced GC. This was implemented in a garbage

collection simulator having trace files as input. Experimental results show that a thread-based approach triggered more collections than the Balanced GC but at a shorter time per GC. On the other hand, the overall simulation time was faster in the Balanced GC. In terms of other factors such as the number of objects copied, the total bytes copied, and object reference updates, thread-based was generally lower, while in terms of remset updates the Balanced GC was lower. In terms of regions freed and bytes freed, Balanced GC was generally higher.

The experimental results between the two approaches are generally similar. The proposed advantage of a thread-based approach over Balanced GC is in the increased parallelism that can occur as a result of being able to collect from a subset of threads instead of the entire set. The experiments on MyThreads suggest that from the point of view of escaping objects and in its absence, garbage collection can be performed on a small set of threads without hindering the continued execution of other threads. On the other hand, benchmarks with high relationships among the threads in terms of escaping objects will need to suspend more threads during GCs thereby rendering pause times similar to the stop-the-world fashion. In summary, the results suggest that for applications with few escaping objects, thread-based GC is more promising than Balanced GC.

As a variation of this study and as possible future work, further investigation of the thread-based approach can be conducted in a multi-threaded environment. Currently, the garbage collection simulator is a single-threaded ap-

plication that reads a single trace file sequentially. Simulating thread-based garbage collection in a multi-threaded setting can bring more accurate results in terms of the timing of GC occurrences which can improve the results in other metrics as well. The simulated concurrent execution of threads can be extended to determine which threads are dependent on other threads in terms of escaping objects, which can be a basis for thread-grouping. Since trace files are usually very large, each thread group can have its own independent trace file as input. Having a shared memory, some of the factors that would be interesting to determine, including, the frequency of garbage collection, the throughput, and the cumulative pause times for each thread group.

Bibliography

[1] Mendel Rosenblum. The reincarnation of virtual machines. *Queue*, 2(5):pp. 34–40, July 2004.

[2] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[3] Manfred Jendrosch, Gerhard Dueck, Charlie Gracie, and André Hinkenjann. PC-based escape analysis in the Java Virtual Machine. In *5th International Conference on Software Technology and Engineering (ICSTE)*, 2013.

[4] Burka Peter Micic Aleksandar Sciampacone, Ryan. Garbage collection in WebSphere Application Server V8, part 2: Balanced garbage collector as a new option. Technical report, IBM Ottawa Lab, 2011.

[5] IBM Corporation. IBM knowledge center: How to coexist with the garbage collector, April 2005.

https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.win.80.doc/diag/understanding/mm_gc_coexist.html.

[6] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):pp. 184–195, April 1960.

[7] Mattias Persson. Java technology, IBM style, Part 1: Garbage collection policies. Technical report, IBM Corporation, 2006.

[8] George Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):pp. 655–657, December 1960.

[9] David Bacon and VT Rajan. Concurrent cycle collection in reference counted systems. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 207–235, London, UK, UK, 2001. Springer-Verlag.

[10] Thomas Christopher. Reference count garbage collection. *Software: Practice and Experience*, 14(6):pp. 503–507, 1984.

[11] Alejandro Martnez, Rosita Wachenchauzer, and Rafael Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34(1):pp. 31 – 35, 1990.

[12] David Bacon, Richard Attanasio, Han Bok Lee, VT Rajan, and Stephen E. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *PLDI*, 2001.

[13] Helena Rodrigues and Richard Jones. Cyclic distributed garbage collection with group merger. In *European Conference on Object-Oriented Programming*, pages 260–284. Springer Berlin Heidelberg, 1998.

[14] Barry Hayes. Using key object opportunism to collect old objects. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 33–46, New York, NY, USA, 1991. ACM.

[15] Urs Hlzle. A fast write barrier for generational garbage collector. In *OOPSLA93 Garbage Collection Workshop*, Washington, D.C., October 1993.

[16] Stephen Blackburn and Kathryn McKinley. In or out?: Putting write barriers in their place. In *ACM SIGPLAN Notices*, volume 38, pages 175–184. ACM, 2002.

[17] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation Journal*, 17:pp. 245–265, 2004.

[18] Peter Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. Technical report (Massachusetts Institute of Technology. Laboratory for Computer Science). Massachussets Institute of Technology, Laboratory for Computer Science, 1977.

[19] Dave Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Softw. Pract. Exper.*, 20(1):pp. 5–12, January 1990.

[20] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. *Region-based Memory Management in Cyclone*, pages 282–293. PLDI '02. ACM, 2002.

[21] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 1–19, New York, NY, USA, 1999. ACM.

[22] Sun Microsystems. Memory management in the java hotspot virtual machine, April 2006. http://www.oracle.com/.../java/javase/memorymanagement-whitepaper-150215.pdf.

[23] Konstantin Nasartschuk, Marcel Dombrowski, Tristan Basa, Mazder Rahman, Kenneth Kent, and Gerhard Dueck. Garcosim: A framework for automated memory management research and evaluation. In *Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS'15, pages 263–268, ICST, Brussels, Belgium, Belgium, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[24] Md Mazder Rahman, Konstantin Nasartschuk, Kenneth B Kent, and Gerhard W Dueck. Trace files for automatic memory management systems. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 2, pages 9–12. IEEE, 2016.

[25] Benjamin Zorn. *Comparative performance evaluation of garbage collection algorithms*. PhD thesis, University of California, Berkeley, 1989.

[26] Stephen Blackburn, Robin Garner, Chris Hoffmann, Asjad Khang, Kathryn McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, and Samuel Guyer. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[27] Kumar Shiv, Kingsum Chow, Yanping Wang, and Dmitry Petrochenko. SPECjvm2008 performance characterization. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 17–35, Berlin, Heidelberg, 2009. Springer-Verlag.

[28] Sasha Goldshtein, Dima Zurbalev, and Ido Flatow. *Pro .NET Performance: Optimize Your C# Applications*. Apress, Berkely, CA, USA, 1st edition, 2012.

[29] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition.* Addison-Wesley Professional, 1st edition, 2014.

[30] Stefan Richthofer. Garbage collection in JyNI - how to bridge mark/sweep and reference counting GC. *CoRR*, abs/1607.00825, 2016.

[31] S. Goldshtein, D. Zurbalev, S. Group, and I. Flatow. *Pro .NET Performance: Optimize Your C# Applications.* Expert's voice in .NET. Apress, 2012.

[32] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *European Conference on Object-Oriented Programming*, pages 92–115. Springer, 1999.

[33] Ran Rinat and Scott Smith. Modular internet programming with cells. In *European Conference on Object-Oriented Programming*, pages 257–280. Springer, 2002.

Appendix A

Benchmark Descriptions

The benchmark suites used were mainly from Dacapo 9.12 and SPECjvm2008. A special purpose Java program was also created to test thread independence in terms of escaping objects.

A.1 Dacapo 9.12

A.1.1 Aurora

Simulates a number of programs run on a grid of AVR microcontrollers.

A.1.2 Batik

Produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik.

A.1.3 Fop

Takes an XSL-FO file, parses it and formats it, generating a PDF file.

A.1.4 Lusearch

Uses Lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible.

A.1.5 Luindex

Uses Lucene to indexes a set of documents; the works of Shakespeare and the King James Bible

A.1.6 Pmd

Analyzes a set of Java classes for a range of source code problems.

A.1.7 Jython

Inteprets a the pybench Python benchmark.

A.1.8 Xalan

Transforms XML documents into HTML.

A.2 SPECjvm2008

A.2.1 Compiler.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. This benchmark uses its own FileManager to deal with memory rather than with disk and file system operations. '-proc:none' option is used to make this benchmark 1.5 compatible.

A.2.2 Compiler.sunflow

This benchmark uses the OpenJDK (JDK 7 alpha) front end compiler to compile a set of .java files. The code compiled is sunflow sub-benchmark from SPECjvm2008. This benchmark uses its own FileManager to deal with memory rather than with disk and file system operations. '-proc:none' option is used to make this benchmark 1.5 compatible.

A.2.3 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.

A.2.4 Sunflow

This benchmark tests graphics visualization using an open source, internally multi-threaded global illumination rendering system. The sunflow library is threaded internally, that is, it is 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, that is, will run as many sunflow benchmark instances in parallel as half the number of hardware threads.

A.2.5 Xml.transform

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.

A.3 MyThreads

A Java program that runs five threads from five different classes. Each thread was tasked to allocate objects to a local variable a number of times to the point of a garbage collection.

Appendix B

Trace File Format

Trace files contain event logs from memory management operations that happened in an instrumented JVM run and are post-processed to the following format to serve as input to the GarCoSim simulator.
The different memory management operations captured in the trace files are as follows:

B.1 Allocation

Most objects allocated by Java applications are short-lived [16]. Given the overhead of allocating an object from the general heap, also known as the *slow-path* allocation, this becomes a bottle-neck in system performance. To circumvent this, performance-oriented JVMs have provisions for assigning regions of memory for exclusive use of a thread. Objects that are identified

as being short-lived are allocated from these exclusive regions, also known as *fast-path* allocation, thereby avoiding the need for expensive synchronization operations. Within both the slow-path and fast-path allocations, we distinguish between single and indexable (arrays) objects. With fast-path indexable objects, we distinguish between contiguous and discontiguous arrays. All in all, we identify five types of allocation. An extra field is added to the j9object to contain the object ID. The integer value assigned to the object id is taken from a global variable that is also added to keep track of the current number of objects allocated.

For every allocation, the information we trace includes the names of threads handling the object allocation, the object id, the size of the object, the number of object reference slots, the class name of the object, the total instance size of the class, and the number of static fields of the class. Format is as follows:

$$a\ T_i\ O_j\ S_k\ N_l\ C_m \tag{B.1}$$

a indicates the allocation operation, T_i is the thread performing the operation, O_j is the object being allocated, S_k is the size of the object, N_l is the number of reference slots that the object will contain, and C_m is the class ID of the object.

B.2 Reference Operation

A reference operation is a way of referencing objects from another object. Objects usually contain reference slots the number of which is indicated during allocation. The format for this operation is as follows:

$$w\ T_i\ P_j\ \#_n\ O_k\ F_m\ S_n\ V_o \tag{B.2}$$

w indicates the reference operation, T_i is the thread performing the operation, P_j is the parent object whose slot $\#_n$ is being assigned to point to object O_k, F_m is the offset of the first slot, S_n is the size of the slot, and V_o indicates whether it's volatile or not.

B.3 Class Operation

A class operation is similar to a reference operation except for the parent object being replaced by a class ID. This implies the slot being assigned is static field of the class. The format for this operation is as follows:

$$c\ T_i\ C_j\ O_k\ S_n \tag{B.3}$$

c indicates the class operation, T_i is the thread performing the operation, C_j is the class whose slot S_n is being assigned to point to object O_k.

B.4 Store Access

Four different store accesses are as follows:

1. **store primitives into an object:** This operation is performed when a variable of an object field is assigned with data of a primitive type. The recorded data are thread name, object id, field size, field offset, and field type (volatile/non-volatile).

2. **store references into an object:** This operation happens when an object references another object. Relevant information needed for this operation includes the object IDs of both the parent object and the child object, and the reference slot number. The JVM has implemented a write barrier whenever fields are going to be stored into. Hooks were added in these barriers to output relevant information such as the thread, the parent and the child object, and the object reference slot where the child object will be pointed from. The format for these store operations are as follows:

$$\text{s } T_i \ P_j \ I_x \ S_n \ V_o \tag{B.4}$$

$$\text{s } T_i \ P_j \ F_m \ S_n \ V_o \tag{B.5}$$

s indicates a store operation, T_i indicates the thread performing the operation, P_j indicates the object to be written into, I_x is the slot

number, or F_m if it's slot offset, S_n is the slot size, and V_o indicates whether the fiels is volatile or not.

3. **store primitives into a class:** This operation is performed when a static variable of a class field is assigned with data of a primitive type. The recorded data are thread name, class name, field size, field offset, field type (volatile/non-volatile).

4. **store references into a class:** This operation similar to object-to-object referencing except that the object reference slot is a static field. The recorded information is thread name, class name, field offset/index, object id, field size, field type (volatile/non-volatile). The format for these store operations are as follows:

$$\text{s } T_i \; C_j \; I_x \; S_n \; V_o \tag{B.6}$$

$$\text{s } T_i \; C_j \; F_m \; S_n \; V_o \tag{B.7}$$

s indicates a store operation, T_i indicates the thread performing the operation, C_j is the class to be written into, I_x is the slot number, or F_m if it's a slot offset, S_n is the slot size, and V_o indicates whether the fiels is volatile or not.

B.5 Read Access

We also instrument an object whenever it (or one of its field) is accessed. Information from this operation can be used for analyzing the effects of caching. Formats are as follows:

$$r\ T_i\ O_j\ I_x\ S_n\ V_o \tag{B.8}$$

$$r\ T_i\ C_j\ I_x\ S_n\ V_o \tag{B.9}$$

r indicates a read operation, T_i is the thread performing the operation, an O_j indicates an object, a C_j indicates the class,whose slot I_x is to be read, S_n is the size of the slot, and V_o indicates whether it is volatile or not.

B.6 Rootset Dump

Rootset dumps are a way of inspecting a thread's rootset. Since this should be done in a *stop-the-world* fashion, an asynchronous-handler is signaled whenever allocation is done. The timing of the actual dump depends entirely on the JVM. Rootset additions and rootset deletions are inferred between two rootset dumps in the next phase. Formats are as follows:

$$+\ T_i\ O_j \tag{B.10}$$

$$-\ T_i\ O_j \tag{B.11}$$

$+/-$ indicates the rootset operation, a "+" for an addition, and "-" for deletion, T_i indicates the thread whose rootset the operation will be performed, and O_j is the target object to be unreferenced.

Appendix C

Source Code

```
public class MyThreads {

    public static void main(String[] args){

        AllocThread1 t1 = new AllocThread1(2500000);

        AllocThread2 t2 = new AllocThread2(1500000);

        AllocThread3 t3 = new AllocThread3(750000);

        AllocThread4 t4 = new AllocThread4(2000000);

        AllocThread5 t5 = new AllocThread5(250000);

        t1.start();

        t2.start();

        t3.start();

        t4.start();

        t5.start();

        try {
```

```
          while (t1.isAlive() && t2.isAlive() && t3.isAlive()
                 && t4.isAlive() && t5.isAlive()) {
                 Thread.sleep(1500);
             }
         }
         catch(InterruptedException e) {
             System.out.println("Main thread interrupted");
         }
     }
}

public class AllocThread# extends Thread {
     int    allocSize;   //how much to allocate
     int    i;
     Object x;
     AllocThread1(int size) {
         allocSize  = size;
     }
     public void run() {
         for (i = 0; i < allocSize; i++)
             x  = new double[1];
     }
}
```

Vita

Name: Tristan M. Basa
Address: 289-2 Regent St., Fredericton, New Brunswick, Canada
Mobile: (506)260-7921
Email: *tbasa@unb.ca*

Education

MS Comp. Sci., University of New Brunswick, Fredericton (continuing).
MS Comp. Sci., University of the Philippines, Diliman, 2000.
B Comp. Sci., University of the Philippines, Diliman, 1995.

Professional Experience

Asst. Professor, Dept. of Comp. Sci., UP Diliman, Philippines, 2008–13.
IT-Consultant, Ayala Mind Museum, Taguig City, Philippines, 2011
IT-Consultant, Summit Realty Inc., San Juan, Philippines, 2003–10.
Lecturer, Kalayaan College, Marikina City, Philippines, 2003–04.
Senior Developer, K2 Interactive Inc., Pasig City, Philippines, 2000–03.
Software Engr., Interim Asia Ltd., Makati City, Philippines,1997–98.
Programmer, Software Brewers Inc., Makati City, Philippines, 1995-96

Selected Publications

GABAY: An e-Learning Filipino Sign Language Tutorial Using Manifold Learning and Dynamic Time Warping, 8^{th} National Conference on Information Technology Education, Boracay, Philippines, 2010, (Best Paper)

Using Psychological Signals to Evolve Art, 4th EvoMusART, LNCS, Budapest, Hungary, 2006.

MMORPG Map Evaluation Using Pedestrian Agents, Christian Anthony L. Go, Tristan M. Basa, Won-Hyung Lee, Proceedings of the 1^{st} International Conference on Advances in Hybrid Information Technology, pp. 323-332, Jeju, South Korea, 2006.

A Bayesian Approach to a Computational Model of Curiosity, KSII Conf., Suwon, South Korea, 2004 (Best Paper)

Skills

C, C++, Java, JavaScript, PHP, ASP, Visual Basic, HTML, MySQL, MS SQL, Postgres, JVM, Linux, Unix, Windows