Concurrent Garbage Collection

Presented to

College of Engineering & Computer Science

In Fulfillment of the Requirements

for COMP8755 Computing Individual Projects

By Zhansong Li (u5844206)

Supervised by Professor Steve Blackburn

Australian National University

The Australian National University

Canberra ACT 2601 Australia

Oct. 2017

Abstract

Garbage collection is an essential strategy for automatic memory management in many modern programming languages. In general, there are two main approaches to garbage collection: reference counting identifies dead objects using the number of incoming references and tracing identifies live objects. After decades of poor performance, a state-of-the-art reference counting collector RCImmix managed to match the performance of highly-tuned production tracing collectors. RCImmix combines Immix heap organisation and optimisations to reference counting from the RC collector. However, these optimisations also require the stop-the-world phases where all mutators have to be stopped at the same time.

The main contribution of this project is a concurrent implementation of RCImmix in JikesRVM that can process part of the stop-the-world collection concurrently with the mutators. Performance comparisons between the original RCImmix and the concurrent version show that the concurrent version managed to reduce median pause time for all benchmarks while achieving similar throughput.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Problem Statement

Two main approaches exist for automatic garbage collection. Reference counting keeps track of incoming references while tracing calculates a transitive closure from mutator roots to identify live objects. Reference counting has a number of advantages, including the ability to immediately reclaim dead objects and object local scope. However, high performance garbage collectors tend to use tracing algorithm since performance of reference counting algorithms lag behind tracing algorithms by 30% [1].

RCImmix [1] is a state-of-the-art reference counting collector that managed to improve performance of reference counting to match a highly-tuned production tracing algorithm GenImmix[1]. These optimisations require the use of a stop-the-world phase where all mutators are stopped at the same time, which can have a significant impact on mutator performance.

1.2 Scope and Contribution

This project aims to reduce the pause time of RCImmix by making some part of the stop-the-world collection run concurrently with the mutators. A concurrent version of the algorithm that can process dec-buf concurrently with the mutators is implemented in JikesRVM 3.1.4[2].

Comparisons of throughput and pause time between the original RCImmix and the concurrent version are performed using DaCapo benchmark [2].

1.3 Outline of the Report

For the rest of the report, Chapter 2 provides some background information about garbage collection in general as well RCImmix in particular. Chapter 3 discuss the buffering and concurrent

[1] http://jikesrvm.sourceforge.net/apidocs/latest/org/mmtk/plan/generational/immix/GenImmix.html
[2] http://www.jikesrvm.org/

dec-buf processing, the main contribution of this project. Chapter 4 lists testing environment as well as experimental results. Finally, Chapter 5 gives a summary of the report and discuss possible future work.

CHAPTER 2

Background and Related Work

This chapter provides some background information on garbage collection and describe the RCImmix collector [1] on which the concurrent collector implemented in this project is based.

2.1 Garbage Collection

This section gives an overview of garbage collection, including the motivation for garbage collection, basic terminologies and two main types of algorithms, reference counting and tracing.

2.1.1 Why Garbage Collection

Garbage collection is a strategy for automatic memory management where objects that are no longer used by a program are automatically reclaimed. It is widely used in modern programming languages such as Java and Python. There are several advantages for garbage collection. First, it makes it easier to write correct code by preventing memory leak caused by not freeing objects or freeing objects multiple times. Another advantage not often mentioned [3] is that garbage collection uncouples the problem of memory management from interface design.

2.1.2 Terminology

Following [3], this section introduces a few terminologies specific to garbage collection research.

- **mutator**: the mutator executes application code, allocates new objects, and mutates the object graph by changing their reference fields.
- **collector**: the collector identifies and reclaims dead objects. Since liveness is undecidable [3], all collectors use reachability to approximate liveness, i.e. if an object is reachable from the program code, then it's considered live.

- **stop-the-world, concurrent and on-the-fly collector**: collectors can be separated into three categories, shown in Figure 2.3. Stop-the-world collectors stop all mutators at the same time resume execution after collection is complete. In concurrent collection, stop-the-world pause is still required, but a large part of the collection process can be done concurrently with the mutator. On-the-fly collection stops one mutator thread at a time, eliminating the need for stop-the-world pauses.
- **parallel collector**: while earlier literature sometimes use parallel and concurrent interchangeably, modern usage of the word is that multiple collector threads work together during collection.
- **epochs and phases**: a collection algorithm is typically divided into different phases, and collection execution is divided into epochs. For stop-the-world and concurrent collection, an epoch is usually the period of time between two stop-the-world pauses.
- **root**: roots or mutator roots are objects stored in static variables, thread stacks as opposed to the heap.

2.1.3 Reference Counting and Tracing

There are two main approaches to garbage collection, reference counting [4] and tracing [5]. In reference counting, dead objects are identified by counting the number of incoming references; in tracing, instead of identifying dead objects, live objects are identified by performing transitive closure over the object graph from roots.

Compared to tracing, simple reference counting has several advantages. It can immediately reclaim dead objects, although this might not be desirable in some cases. It can also be implemented without assistance from or knowledge of the run-time system. It's object-local, that is, during collection it only needs to look at objects that are changed by the program instead of all objects on the heap. This can be shown in Figure 2.4 and Figure 2.5, where reference counting only needs to consider dead objects F and G, and the tracing algorithm need to look at all live objects.

There are also some disadvantages. First, in order to avoid race conditions during updates of reference counts, read and write barriers for pointer load and restore need to be added, where as simple tracing algorithms don't require mutator barriers at all. Second, since most objects are small [2], space overhead for storing refcount can be significant. Third, simple reference counting can't collect cycles. Finally, due to manipulation of reference counts, it can turns read-only operations such as array traversal into ones that require writes to memory, thereby polluting cache [3].

The next section will show how optimisations included in RCImmix can solve some of these problems.

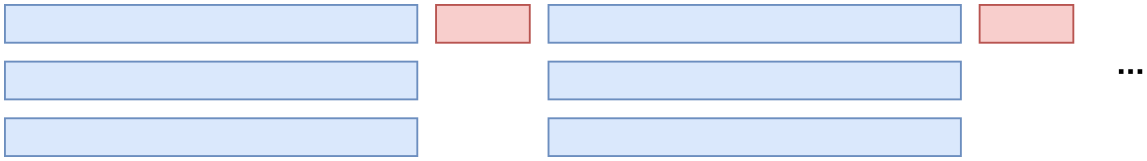

FIGURE 2.1 In stop-the-world collection, all mutators are stopped at the same time and resume execution after collection is complete.

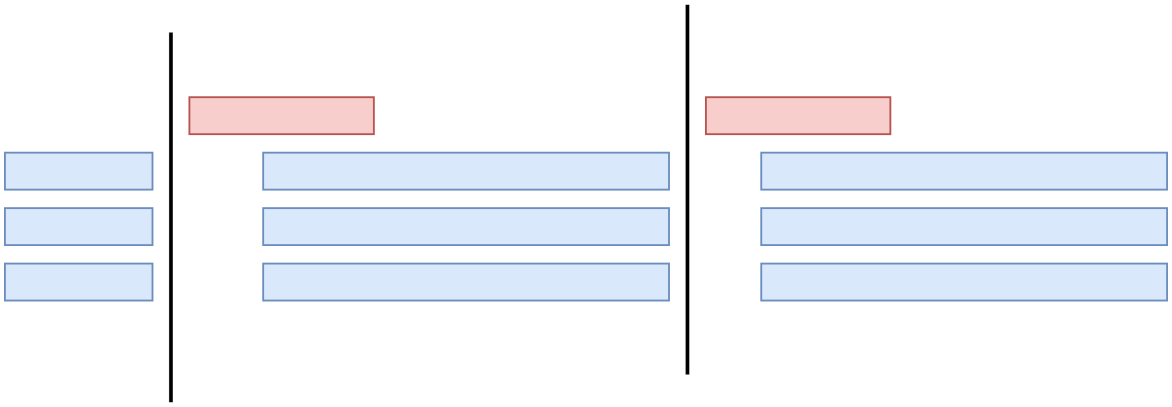

FIGURE 2.2 In concurrent collection, stop-the-world is still required but a large part of the collection process can be done concurrently with the mutator.

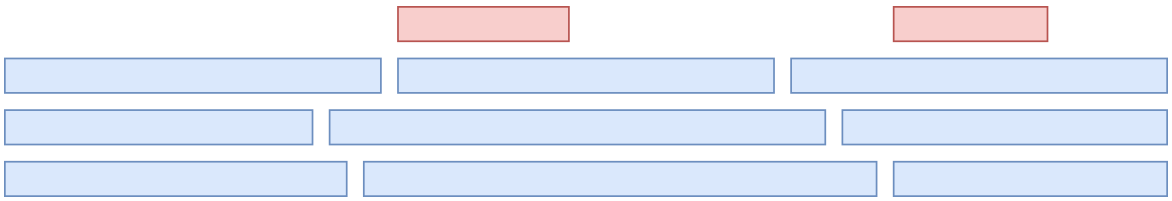

FIGURE 2.3 On-the-fly collection stops one mutator thread at a time, eliminating the need for stop-the-world pauses.

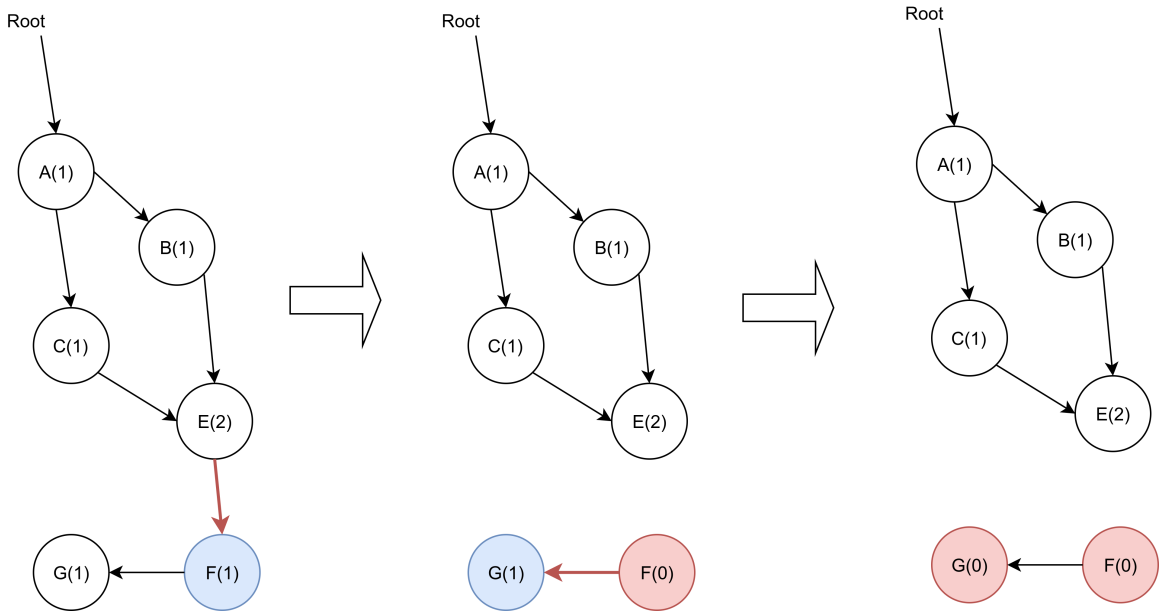

FIGURE 2.4 Simple reference count. The mutator removes the reference from E to F, reducing the reference count of E to zero, triggering collection. The children of E are processed recursively, and the reference count of F is reduced to zero as well. Object E and F are reclaimed.

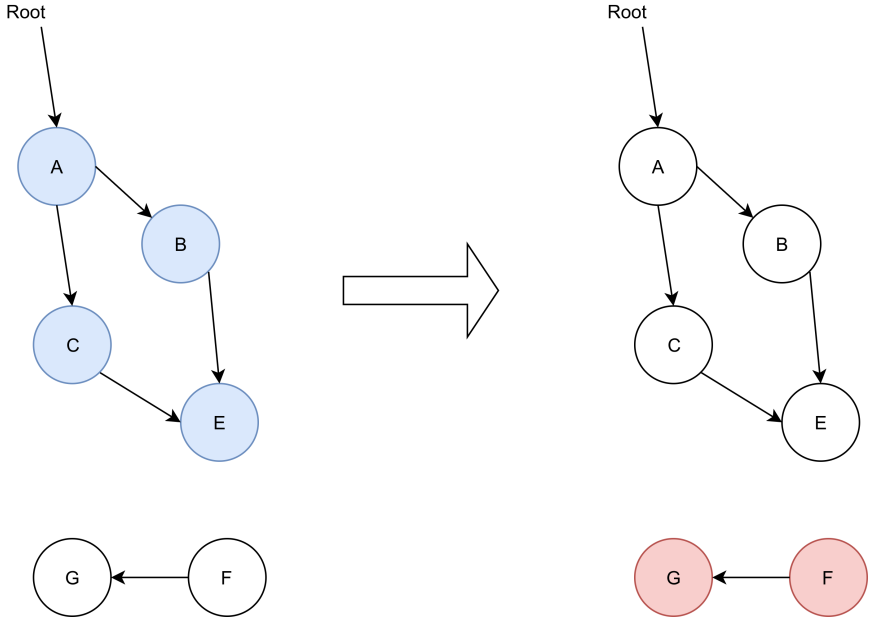

FIGURE 2.5 Simple mark-and-sweep tracing. During collection, all objects reachable from the mutator roots (A, B, C, and E) are marked and during sweeping all unreachable objects (F and G) are reclaimed.

2.2 RCImmix

RCImmix combines RC reference counting algorithm [6] with Immix heap organisation [7]. The following sections will look at optimisations for reference counting included in RCImmix.

2.2.1 Optimisations to Reference Counting

2.2.1.1 Limited Bits Reference Counting

Most objects have small refcounts. According to [1], the refcounts for 99.35% of the objects allocated can be represented using only three refcount bits. Consequently, RCImmix is able to fit all information needed for collection in a single object header byte. For objects with reference counts that can't be fit in three bits, their reference counts are considered 'stuck' and won't be changed or collected, this introduces a potential memory leak. In RCImmix, this is solved by using the backup cycle collection tracer to reset and correct all reference counts. Shahriyar et al [1, 6] shows that this strategy works well.

2.2.1.2 Cycle Collection

One disadvantage for refcount algorithms is that they can't collect cycles, but cycles are common in many data structures such as doubly linked lists. To ensure completeness, the simplest approach, which is also adopted by RCImmix, is to use a backup tracing collector that occasionally performs whole-heap mark-and-sweep. Alternatively, partial tracing algorithms based on trial-deletion such as the recycler [8] can be used.

2.2.1.3 Deferred Reference Counting

Manipulating reference counts can be expensive compared to tracing algorithms. Given that the vast majority of pointer loads are to registers and stack [3], e.g. temporary and local variables, one optimisation introduced by [9] defers the counting of these objects to when they are stored into the heap. With deferred reference counts the reference counts are no longer accurate, so when they drop to zero these objects are instead added to a zero count table (ZCT). During stop-the-world collection the collector will correct these reference counts and reclaim any non-root objects in the ZCT.

RC uses a buffering technique introduced by [8] instead of the ZCT. During stop-the-world phases the reference counts for objects in the root set are temporarily increased and then decrements for these objects are queued for the next epoch.

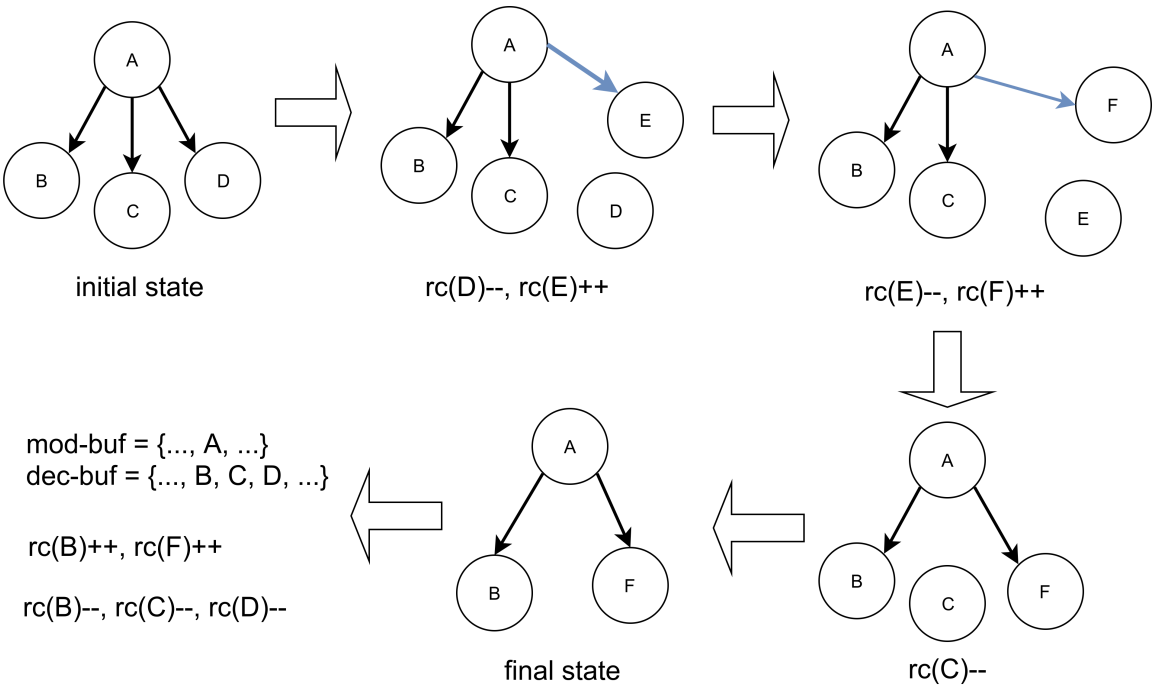

FIGURE 2.6 Illustration of coalesced reference counting. The updates to the reference count of E are coalesced.

2.2.1.4 Coalesced Reference Counting

Levanoni and Petrank [10] observed that for a given period, only object fields at the start and end need to be considered in order to ensure correct reference counts and all intermediate operations can be safely ignored. An example is shown in Figure 2.6.

This is implemented using an object remembering mutator write barrier. Each object contains a log bit which records whether it has been modified before the current epoch. When an unlogged object is first modified, the original object fields are added to a dec-buf and the object is added to a mod-buf and the log bit of the object is set. Later changes to the object fields are ignored. During collection, the reference counts for objects referenced by logged objects, which effectively captures a snapshot of the heap at the beginning of the current epoch, are increased. The reference counts for objects in the dec-buf, which captures a snapshot of the heap at the end of the previous collection, are decreased. These operations in effect replays and coalesces all reference count updates by the mutator.

2.2.1.5 Young Objects

According to the weak generational hypothesis, most objects die young [3]. This is exploited in most modern garbage collection implementations, including Oracle's Hotspot VM[1] and

[1]http://www.oracle.com/technetwork/java/whitepaper-135217.html

JikesRVM's production collector GenImmix[2], by using different collection plans or policies for young and old objects e.g. with a generational nursery.

In RC, two optimisations are used to more effectively deal with young objects: lazy mod-buf insertion and allocate as dead, which are part of the same strategy. New objects are allocated with zero reference count and their new bits are set in the object headers. They are only added to mod-buf when the collector performs the first increment which also clears their new bit. Consequently, during Immix sweeping, to be discussed in Section 2.2.2.2, new objects not reachable from old objects and the root are automatically collected.

2.2.2 Heap Organisation

This section will discuss heap organisation, i.e. the management of allocation and reclamation of objects on the heap.

2.2.2.1 Free-List

Free-list allocators divide the heap into free cells of memory. When an allocation request is received, the allocator finds a sufficiently large cell to put the newly allocated buffer. Different policies exist for finding which free cell to satisfy the request, such as first-fit, best-fit, and next-fit, with different trade-off between allocation time and level of fragmentation. Modern free-list implementations typically use size-segregated free lists [3], which allocate objects of different size in different free lists with cells of fixed sizes, called size classes. This can reduce both fragmentation and allocation time.

Support for immediate and fast reclamation of objects makes segregated free lists suitable for reference counting. They are also easy to sweep because information about free/occupied memory is stored in a separate data structure. However, free lists have several drawbacks. First, internal fragmentation can occur when objects are not matched perfectly to one of the size classes. Second,external fragmentation can also happen when small free cells fail to satisfy the requirements for allocation and are left unused. Lastly, because free lists often place contemporary objects in different locations, e.g. if they have different sizes, they can cause poor locality for mutator memory access.

2.2.2.2 Immix

The Immix collector uses a mark-region system where the heap is divided into lines and blocks, which make use of cache and page locality. An illustration of Immix heap organisation can be found in Figure 2.7.

[2]http://jikesrvm.sourceforge.net/apidocs/latest/org/mmtk/plan/generational/immix/GenImmix.html

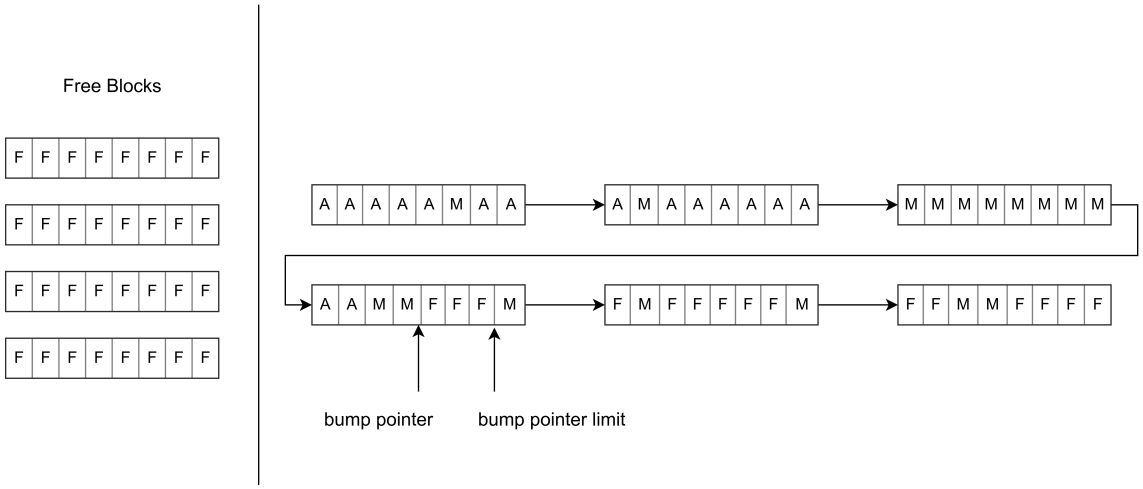

FIGURE 2.7 Illustration of Immix heap organisation. 'A' represents freshly allocated lines, 'F' represents free lines, and 'M' represents lines marked in previous collection.

In Immix space, Objects may span lines but not blocks. Immix uses a simple bump pointer for allocation: the pointer skips over occupied lines and place the object in the first available line(s). When no blocks are available a free block allocator returns a new block. This solves the locality problem for free-list since contemporary objects will likely be allocated in the same block in adjacent lines. Lines have reference counts that keep track of the number of objects contained and blocks are reclaimed during sweeping where blocks that contain no used lines are returned to the global free block allocator.

Since Immix can suffer from fragmentation when a few lines might keep an entire block unavailable for reuse, opportunistic copying is used to reduce fragmentation. During sweeping fragmentation statistics is collected for each block and the most fragmented blocks are flagged as candidate, or source blocks, for compaction. Based on these statistics the allocator reserves target blocks to which lines from the source blocks are copied to. Since these statistics are not exact, if there is not enough room defragmentation is stopped early.

2.3 Summary

This chapter gave an overview of garbage collection and the RCImmix collector. The following chapter will describe buffering and the concurrent snapshot algorithm in more detail.

CHAPTER 3

Concurrent Snapshot Algorithm

This chapter gives describes in more detail buffering in RCImmix [1] and the main contribution of this project, the concurrent snapshot algorithm.

3.1 Buffering

This section describes buffering in RCImmix. The main idea behind buffering is to delay the manipulation of reference counts by capturing a snapshot of changes in heap objects. Pseudocode for the algorithm can be found in Appendix A.

Two buffers, mod-buf and dec-buf, are used to record changes in the object graph. Mod-buf records objects whose fields, i.e. outcoming references, are changed, and dec-buf records the object fields before the change. An object remembering barrier for the mutators needs to be added in order to add objects and their fields to mod-buf and dec-buf. In addition, in order to avoid logging the same object multiple times, the object header needs to record whether the object has been logged or not.

RCImmix uses buffering to implement both deferred and coalescing reference count. Two global shared double-ended queues are used to store logged objects and previous references in object fields. To reduce synchronisation cost threads maintain local buffers in thread-local storage and will only add references to the global queue when the local buffers are full.

An illustration of buffering in RCImmix is shown in Figure 3.1. In RCImmix both mod-buf and dec-buf are processed in stop-the-world phases. The next section will discuss how to move dec-buf processing to a concurrent phase.

3.2 Concurrent dec-buf Processing

This section discusses how to move dec-buf to concurrent phase. Pseudocode for the algorithm can be found in Appendix B.

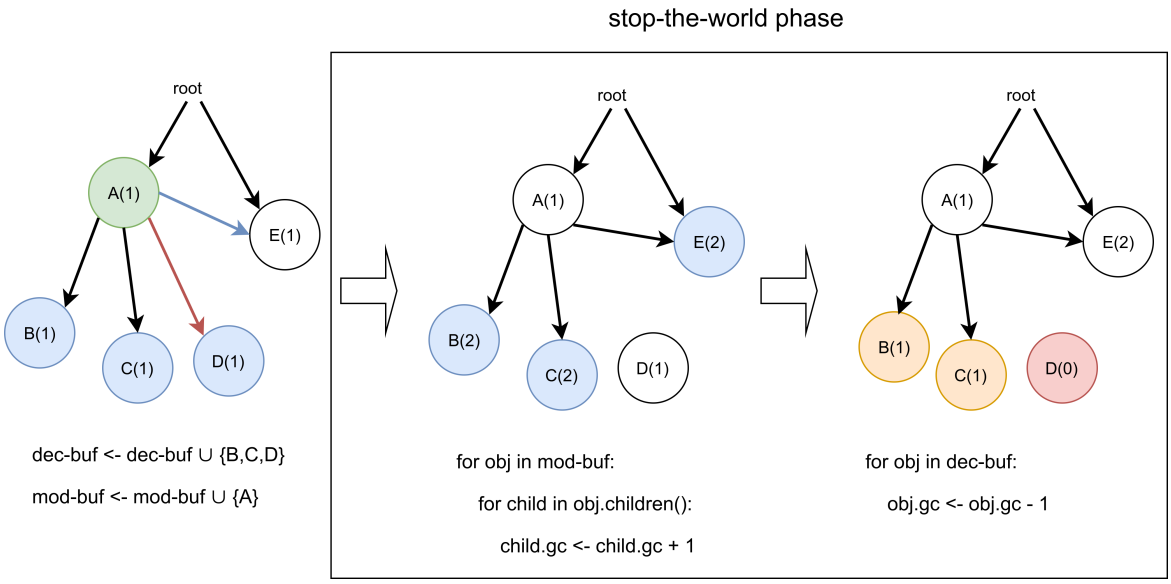

FIGURE 3.1 RCImmix buffered reference counting. Updates to reference counts are replayed and coalesced during stop-the-world collection and correct reference counts are maintained.

In buffered reference counting, the mutators only change the logged bits of the object header, so there is no conflict with the manipulation of reference counts during mod-buf and dec-buf processing during collection. Given that mod-buf and dec-buf in effect capture a snapshot of changes to objects by the mutator between two collection epochs, following the snapshot algorithm in [10], ending the stop-the-world collection early and processing reference count decrements in dec-buf concurrently will not affect the correctness of reference counts.

However, it's important to note that the decrements from the previous epoch and the current epoch must be separated since the correctness of buffering relies on processing mod-buf before dec-buf. As a counter-example, consider an object with reference count of one that is unchanged by the mutator. The parent of the object is logged and the object is added to the dec-buf. If the object decrement is processed before the start of the next epoch, the object will have zero reference count and the lines marked unused, which is clearly incorrect. If mod-buf is processed before dec-buf, its reference count will be increased first, and then decreased later. This results in the correct count.

To separate decrements between two epochs, two global queues and two dec-bufs need to be used. While the mutator is adding references to the current dec-buf, the concurrent collector can work through queued decrements from the previous epoch in another dec-buf. Using two global queues and dec-bufs ensure clear separation of dec-buf processing between epochs.

An illustration of buffering in RCImmix is shown in Figure 3.2. The processing of dec-buf has been moved to a concurrent phase.

One issue with the current implementation is that the blocks that might be freed from concurrent dec-buf processing are not reclaimed until the end of the next epoch. This might affect the perfor-

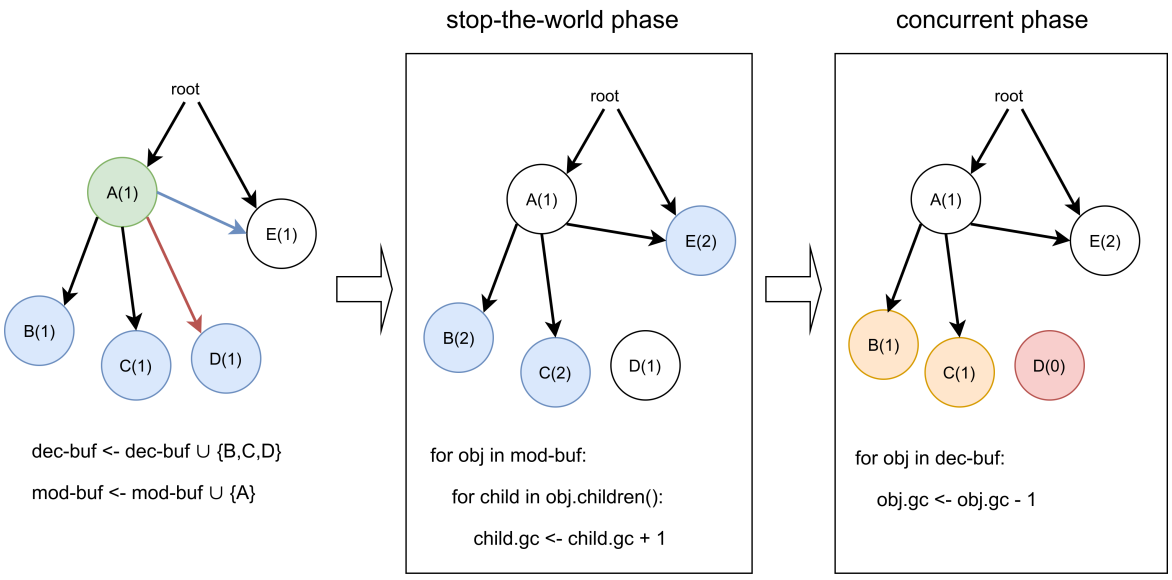

FIGURE 3.2 Concurrent dec-buf processing. Processing of dec-buf has been moved to a concurrent phase.

mance of RCImmix by influencing factors such as frequency of backup tracing and the statistics gathering for Immix opportunistic fragmentation.

3.3 Implementation

The original RCImmix was implemented using the JikesRVM MMTk framework[1]. The concurrent implementation is based on an updated version of RCImmix using the current 3.1.4 release of JikesRVM on GitHub[2].

3.4 Summary

This chapter describes buffering in RCImmix and the concurrent snapshot algorithm. The following chapter will compare benchmark performance between RCImmix and the concurrent version.

[1] http://www.jikesrvm.org/UserGuide/MMTk/index.html
[2] https://github.com/rifatshahriyar/JikesRVM-3.1.4

CHAPTER 4

Results

This chapter describes the benchmark environment and compares results between RCImmix [1] and the concurrent algorithm.

4.1 Testing Environment

TABLE 4.1 Hardware platform for the testing environment

Platform	Clock Frequency	Processors/Threads	L2 Cache	RAM
Intel (R) Core (TM) i7-4770	3.4GHz	4/8	8MB	8GB

The hardware specification for the testing machine can be found in Table 4.1. The operating system is Ubuntu 16.04.2 LTS, with Linux kernel version 3.13.0-88 64-bit (x86_64). Eight benchmarks from the DaCapo benchmark [2] version 2006-10-MR2[1] were chosen.

Each benchmark was run 20 times, and for each run, the first four iterations were used for warmup and the results for the fifth iteration was collected. Compiler replay[2] was used to eliminate variation in memory allocation and non-determinism of the adaptive compiler. A single GC thread is used for all experiments.

4.2 Benchmark Results and Discussion

Two metrics, pause time and throughput, are measured for the original RCImmix and the concurrent version using the MMTk callback interface[3]. For throughput, total time, garbage collection time and mutator time are measured. Heap size for each benchmark is set to twice the minimum heap size required for RCImmix, which can be found in [1]. For pause time, heap size is set to 1G

[1] available at http://www.dacapobench.org/
[2] details can be found at http://www.jikesrvm.org/UserGuide/ExperimentalGuidelines/index.html#x8-640006.2
[3] details can be found in http://www.jikesrvm.org/UserGuide/ExperimentalGuidelines/index.html#x8-680006.3

and GC is controlled using stress factor, which determines the frequency of garbage collection. For instance for a stress factor of 8M, the VM will trigger garbage collection for every 8M of memory allocation. The smaller the stress factor the more frequent garbage collection will occur. Two stress factors, 40M and 8M, are used to compare pause time with both infrequent and frequent garbage collection.

The results for throughput are shown in Table 4.2, Figure 4.1, Figure 4.2, and Figure 4.3. The percentages in the bar plots show slowdown of the concurrent version. For instance, 5% means the concurrent version is 5% slower and -2% means the concurrent version is 2% faster. The results show that except for luindex, the concurrent version performed worse in both mutator time and GC time. Overall, as shown in 4.2 the concurrent version is 2% slower in mutator and total time and 22% slower in GC time. The reason for this is unclear but might be due to interactions between the concurrent phase and the rest of the algorithm, in particular Immix block sweeping, back up tracing, and defragmentation. More experiments are needed to explore these possibilities.

The results for pause time shown from Figure 4.4 and Figure 4.5. The yellow line in boxes represent median pause time and the upper and lower side of the boxes represent the first and third quartiles. The ends of whiskers represent 5% and 95% quantiles for pause time. The results indicate that median pause times for all benchmarks are decreased for both frequent and infrequent garbage collection.

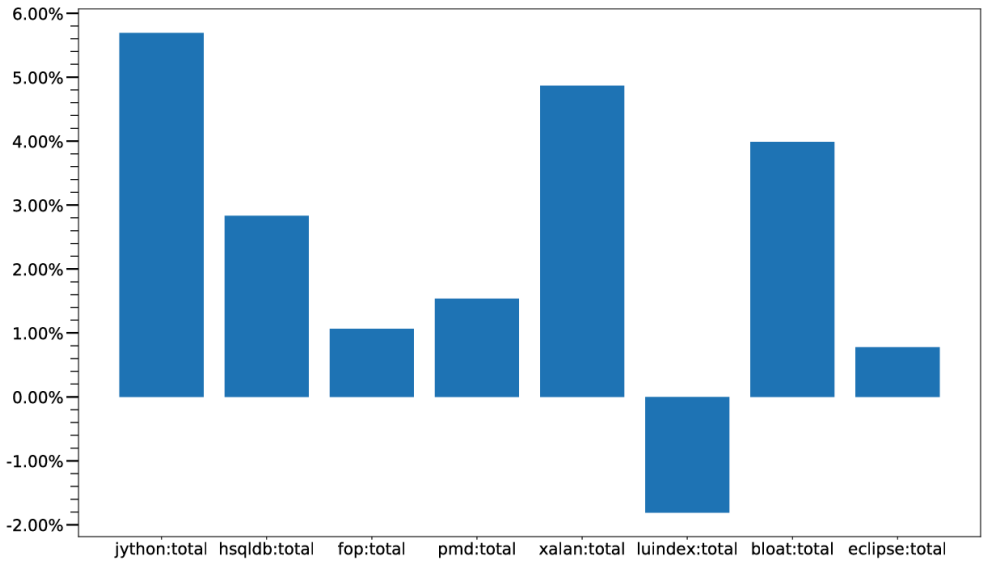

FIGURE 4.1 Comparison of run time between RCImmix and concurrent RCImmix

TABLE 4.2 Comparison of runtime between RCImmix and concurrent RCImmix, time in milliseconds. Intervals are derived from t-test with 95% confidence.

Benchmark	RCImmix			concurrent			concurrent/RCImmix		
	Mutator	Collector	Total	Mutator	Collector	Total	Mutator	Collector	Total
jython	1881.35 ± 6.63	14.41 ± 0.48	1895.76 ± 6.94	1973.19 ± 8.10	30.49 ± 0.71	2003.68 ± 8.66	1.05	2.12	1.06
hsqldb	832.50 ± 30.53	197.23 ± 8.74	1029.73 ± 33.43	870.90 ± 35.17	187.99 ± 2.01	1058.89 ± 36.17	1.05	0.95	1.03
fop	609.09 ± 1.93	15.92 ± 0.29	625.01 ± 1.82	614.33 ± 2.09	17.35 ± 0.35	631.68 ± 2.02	1.01	1.09	1.01
pmd	1655.53 ± 13.10	233.51 ± 16.17	1889.05 ± 28.81	1737.34 ± 36.33	180.70 ± 8.06	1918.04 ± 41.47	1.05	0.77	1.02
xalan	624.65 ± 4.47	365.45 ± 7.13	990.09 ± 8.70	636.03 ± 6.56	402.26 ± 6.12	1038.28 ± 9.47	1.02	1.10	1.05
luindex	2487.21 ± 6.66	11.45 ± 0.36	2498.66 ± 6.73	2430.14 ± 14.22	23.30 ± 1.38	2453.44 ± 14.47	0.98	2.03	0.98
bloat	2513.74 ± 32.61	53.66 ± 3.30	2567.40 ± 34.59	2593.66 ± 18.87	76.09 ± 2.87	2669.75 ± 17.72	1.03	1.42	1.04
eclipse	11441.88 ± 54.54	479.94 ± 13.19	11921.82 ± 62.72	11571.78 ± 58.19	442.50 ± 21.87	12014.28 ± 68.17	1.01	0.92	1.01
geomean							1.02	1.22	1.02

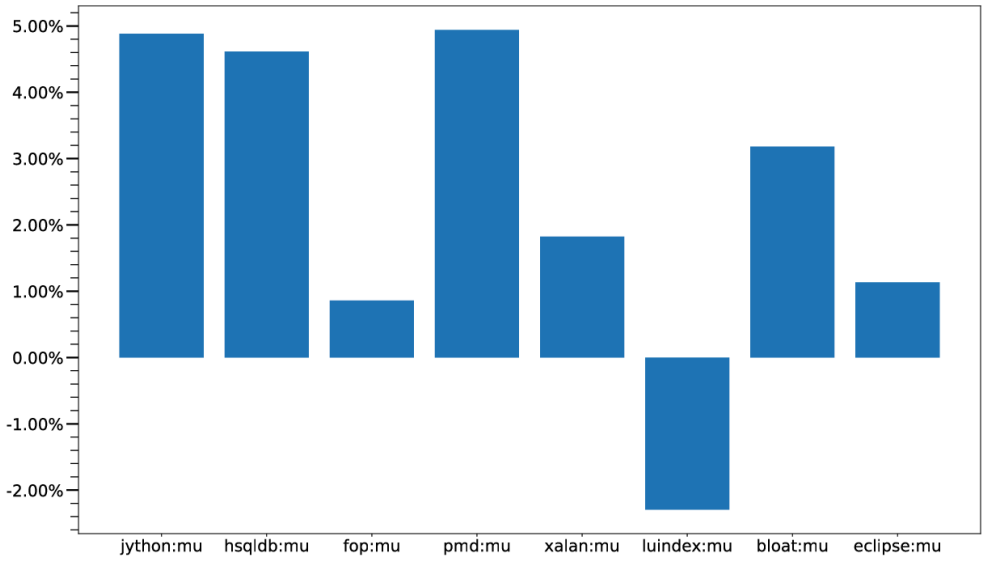

FIGURE 4.2 Comparison of mutator time between RCImmix and concurrent RCImmix

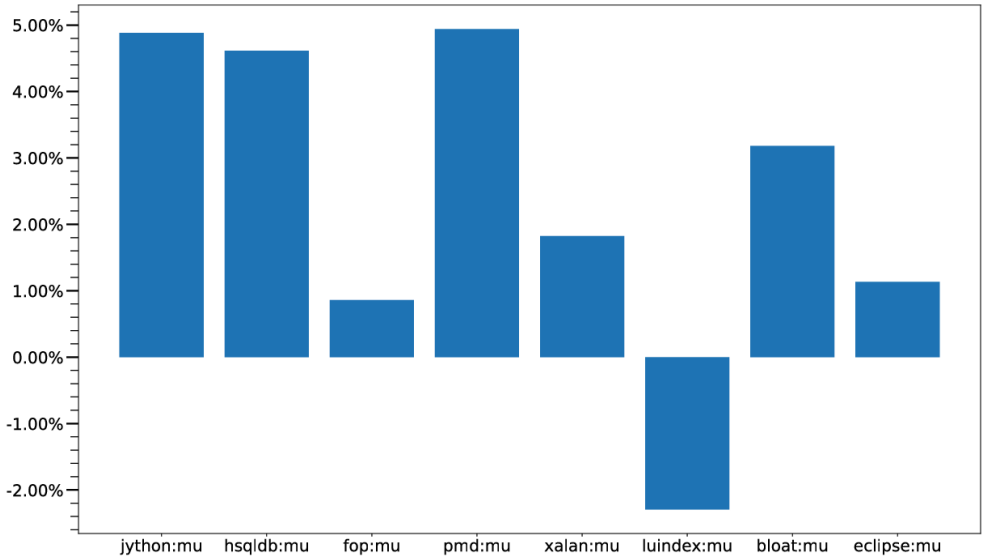

FIGURE 4.3 Comparison of GC time between RCImmix and concurrent RCImmix

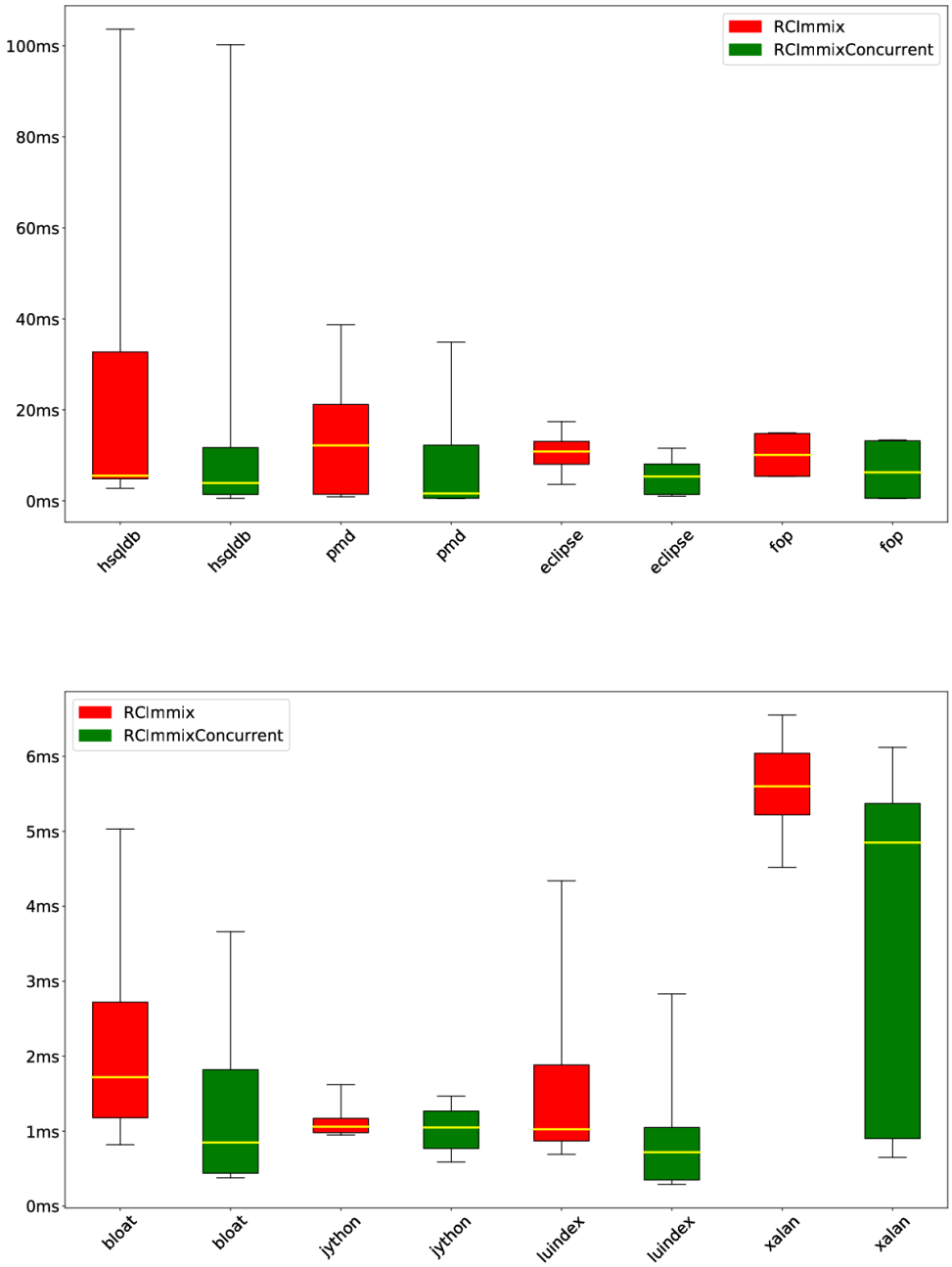

FIGURE 4.4 Comparison of pause time RCImmix and concurrent RCImmix when stress factor is 40M. The yellow lines inside boxes represent median pause times.

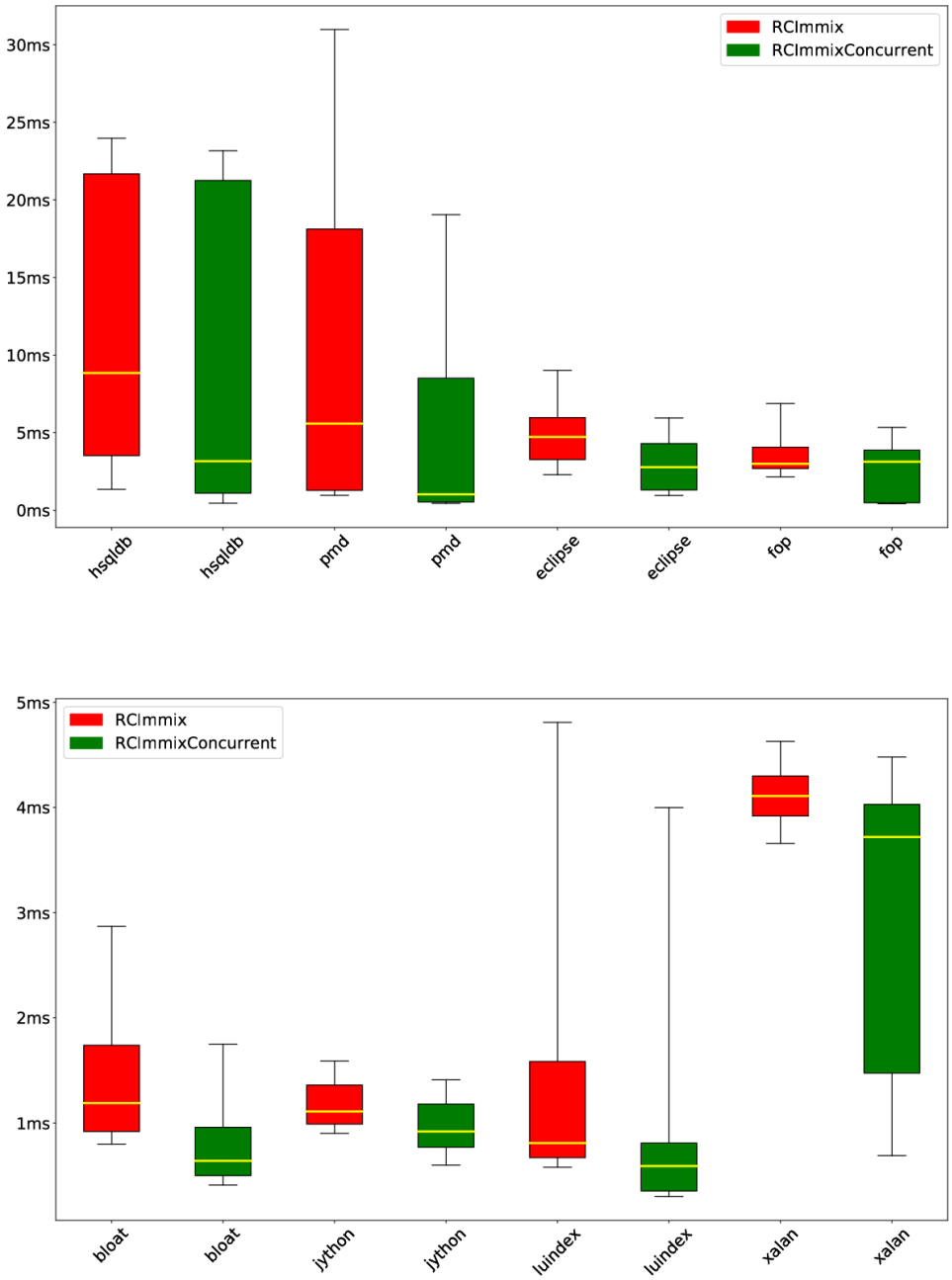

FIGURE 4.5 Comparison of pause time RCImmix and concurrent RCImmix when stress factor is 8M. The yellow lines inside boxes represent median pause times.

CHAPTER 5

Conclusion and Future Work

5.1 Conclusion

RCImmix [1] is a reference counting collector that combines the advantages of reference counting with optimisations from RC [6] and Immix [7] heap organisation. It can match the performance of a production tracing collector GenImmix[1]. Despite its high performance, RCImmix requires stop-the-world pauses to process buffered reference count increments and decrements.

This report describes the design and implementation of a successful concurrent implementation of RCImmix. Performance comparisons show that concurrent RCImmix managed to reduce median pause time for all benchmarks.

5.2 Future Work

Despite its success in reducing pause time, concurrent RCImmix still falls behind the original RCImmix collector in total throughput by 2%. In particular, garbage collection time is 22% worse. In the future, the interaction of the concurrent phase with the rest of the RCImmix collector, especially Immix space and backup tracing, could be investigated further to track down the source of this inefficiency.

At the moment during the concurrent phase if an Immix block becomes reusable it will be released and added to the free blocks allocator at the end of the next epoch during Immix sweep. To improve space utilisation, it's possible to perform Immix sweep at the end of the concurrent phase. However, care must be taken to ensure there is no race between the allocator and the sweep, and if there is a race it's benign one, i.e. incorrect behaviour such as freeing a used block won't happen.

To reduce pause time further, it's possible to make the back up tracing algorithm concurrent as well. The garbage collection handbook [3] discusses several possible algorithms in Chapter 15

[1] http://jikesrvm.sourceforge.net/apidocs/latest/org/mmtk/plan/generational/immix/GenImmix.html

and 16. Making the collector parallel is also likely going to reduce pause time since work during stop-the-world collection could be done faster.

Acknowledgments

This project is completed with the help of Prof. Steve Blackburn, Dr Rifat Shahriyar and Prof. Tony Hosking. I would like to thank Steve for his continued support and help with the design, implementation and evaluation of the concurrent collector. I also want to thank Rifat for help with his RCImmix code. I thank Tony for his excellent textbook on garbage collection. Finally, I would like to thank the programming languages research group at ANU for letting me use their machines to run the benchmarks.

Bibliography

[1] R. Shahriyar, S. M. Blackburn, X. Yang, and K. S. McKinley, "Taking off the gloves with reference counting immix," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 93–110, 2013.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[3] R. Jones, A. Hosking, and E. Moss, "The garbage collection handbook: The art of automatic memory management," 2011.

[4] G. E. Collins, "A method for overlapping and erasure of lists," *Commun. ACM*, vol. 3, no. 12, pp. 655–657, Dec. 1960. [Online]. Available: http://doi.acm.org/10.1145/367487.367501

[5] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[6] R. Shahriyar, S. M. Blackburn, and D. Frampton, "Down for the count? getting reference counting back in the ring," *ACM SIGPLAN Notices*, vol. 47, no. 11, pp. 73–84, 2013.

[7] S. M. Blackburn and K. S. McKinley, "Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2008.

[8] D. F. Bacon and V. T. Rajan, "Concurrent cycle collection in reference counted systems," in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ser. ECOOP '01. London, UK, UK: Springer-Verlag, 2001, pp. 207–235. [Online]. Available: http://dl.acm.org/citation.cfm?id=646158.680003

[9] L. P. Deutsch and D. G. Bobrow, “An efficient, incremental, automatic garbage collector,” *Commun. ACM*, vol. 19, no. 9, pp. 522–526, Sep. 1976. [Online]. Available: http://doi.acm.org/10.1145/360336.360345

[10] Y. Levanoni and E. Petrank, “An on-the-fly reference counting garbage collector for java,” *ACM SIGPLAN Notices*, vol. 36, no. 11, pp. 367–380, 2001.

CHAPTER A

RCImmix buffered reference counting

```
 1: global modPool: global shared queue for mod-buf
 2: global decPool: global shared queue for dec-buf
 3: global oldRoots: root objects for previous epoch
 4:
 5: local modBuf[]: thread local buffers
 6: local decBuf[]: thread local buffers
 7:
 8: function Write(src, i, ref)                          ▷ Mutator write barrier
 9:     if not isLogged(src) then
10:         WriteBarrierSlow(src)
11:     end if
12:     src[i] ← ref
13: end function
14:
15: @atomic                  ▷ Ensure mutual exclusion to avoid logging the same object multiple times
16: function WriteBarrierSlow(src)
17:     modBuf[threadId] ← modBuf[threadId] ∪ { src }
18:     for i ∈ src.fields do
19:         decBuf[threadId] ← decBuf[threadId] ∪ { src[i] }
20:     end for
21:     makeLogged(src)
22: end function
23:
24: function Collect()                                   ▷ Stop-the-World Collection
25:     for ref ∈ oldRoots do          ▷ Queue decrements for the root set of the previous epoch
26:         decBuf[threadId] ← decBuf[threadId] ∪ { ref }
27:     end for
28:     oldRoots ← ∅
29:     for ref ∈ Roots do
```

```
30:         rc(ref) ← rc(ref) + 1                                   ▷ Temporary increment for root objects
31:         if isNew(ref) then
32:             clearNewBit(ref)
33:             ImmixSpace.incLines(ref)                ▷ Ensure new object survives Immix sweeping
34:             modBuf[threadId] ← modBuf[threadId] ∪ { ref }        ▷ Necessary due to allocate as dead
35:         end if
36:         oldRoots ← oldRoots ∪ { ref }                           ▷ To be used in the next epoch
37:     end for
38:
39:     flushAllBuffer(modBuf[], modPool)
40:     flushAllBuffer(decBuf[], decPool)
41:     processModBuf()
42:     processDecBuf()
43:
44:     ImmixSpace.sweepAllBlocks()                                 ▷ Reclaim space
45: end function
46:
47: function processModBuf()
48:     for src ∈ loadToBuffer(modBuf[threadId], modPool) do        ▷ Load all references from global
    modPool to local modBuf
49:         makeUnlogged(src)
50:         for i ∈ src.fields do
51:             rc(src[i]) ← rc(src[i]) + 1
52:             if isNew(src[i]) then                               ▷ Necessary due to allocate as dead
53:                 ImmixSpace.incLines(src[i])         ▷ Ensure new object survives Immix sweeping
54:                 modBuf[threadId] ← modBuf[threadId] ∪ { src }
55:             end if
56:         end for
57:     end for
58: end function
59:
60: function processDecBuf()
61:     for ref ∈ loadToBuffer(decBuf[threadId], decPool) do    ▷ Load all references from global decPool
    to local decBuf
62:         rc(ref) ← rc(ref) - 1
63:         if rc(ref) = 0 then
64:             for child ∈ ref.fields do
65:                 decBuf[threadId] ← decBuf[threadId] ∪ { child }
66:             end for
```

```
67:                 ImmixSpace.decLines(ref)
68:             end if
69:         end for
70:     end function
```

CHAPTER B

Concurrent dec-buf Processing

```
 1: global modPool: global shared queue for mod-buf
 2: global decPool[0..1]: global shared queue for dec-buf
 3: global oldRoots: root objects for previous epoch
 4: global currentDecPool ← 0: current active decPool
 5:
 6: local modBuf[]: thread local buffers
 7: local decBuf[][0..1]: thread local buffers
 8:
 9: @atomic                          ▷ Ensure mutual exclusion to avoid logging the same object multiple times
10: function WriteBarrierSlow(src)
11:     modBuf[threadId] ← modBuf[threadId] ∪ { src }
12:     for i ∈ src.fields do
13:         decBuf[threadId][currentDecPool] ← decBuf[threadId][currentDecPool] ∪ { src[i] }
14:     end for
15:     makeLogged(src)
16: end function
17:
18: function Collect()                                          ▷ Stop-the-World collection
19:     …                                                       ▷ Same as the original RCImmix
20:
21:     flushAllBuffer(modBuf[], modPool)
22:     flushAllBuffer(decBuf[][currentDecPool], decPool[currentDecPool])
23:     processModBuf()
24:
25:     ImmixSpace.sweepAllBlocks()                             ▷ Reclaim space
26:
27:     currentDecPool ← 1-currentDecPool
28: end function
29:
```

```
30: function ConcurrentCollect()                          ▷ Runs concurrently with the mutators
31:     processDecBufConcurrent()
32: end function
33:
34: function processDecBufConcurrent()                    ▷ Work on the decPool for the previous epoch
35:     for ref ∈ loadToBuffer(decBuf[threadId][1-currentDecPool], decPool[1-currentDecPool]) do
36:         rc(ref) ← rc(ref) - 1
37:         if rc(ref) = 0 then
38:             for child ∈ ref.fields do
39:                 decBuf[threadId][1-currentDecPool] ← decBuf[threadId][1-currentDecPool] ∪ { child }
40:             end for
41:             ImmixSpace.decLines(ref)
42:         end if
43:     end for
44: end function
```