2016

Concurrent Compaction in JVM Garbage Collection

Jacob P. Opdahl
University of Minnesota, Morris

Follow this and additional works at: http://digitalcommons.morris.umn.edu/horizons

Part of the OS and Networks Commons

Concurrent Compaction in JVM Garbage Collection

Jacob P. Opdahl
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
opdah023@morris.umn.edu

ABSTRACT

This paper provides a brief overview of both garbage collection (GC) of memory and parallel processing. We then cover how parallel processing applies to GC. Specifically, these concepts are focused within the context of the Java Virtual Machine (JVM). With that foundation, we look at various algorithms that perform compaction of fragmented memory during the GC process. These algorithms are designed to run concurrent to the application running. Such concurrently compacting GC behavior stems from a desire to reduce "stop-the-world" pauses of an application.

Keywords

Garbage Collection (GC), Concurrency, Compaction, Continuously Concurrent Compacting Collector (C4), Field Pinning Protocol (FPP), Collie

1. INTRODUCTION

In programming languages, the allocation and deallocation of memory for objects can be explicit or implicit. If done implicitly, a language is said to have *automatic memory management.* Some languages with automatic memory management are C#, Java, and Python. Use of automatic memory management is beneficial to programmers as it removes their need to worry about the details of memory allocation and deallocation. However, programmers do not have control over how memory management occurs, which can have negative impacts on application performance.

Memory for a program is not an unlimited resource. Automatic memory management must remove unused objects from memory when appropriate. Dead objects, or *garbage*, are objects that can be shown to be unreachable by the program [5]. Garbage should be deallocated to save space for new objects that will be created. The algorithm used to perform implicit deallocation of garbage, by detecting and removing dead objects, is a *garbage collector*. We will focus on garbage collection (GC) performed within Java Virtual Ma-

chines (JVMs), software processes that run programs written in Java and other languages on computing systems [5].

GC is not without cost. Just like the application, GC requires processing resources to run. When using a single core of a processor, we get a *serial* garbage collector operating in a *stop-the-world* fashion [5]; GC requires pausing the application in order to clean up. Growing storage media is leading to more memory availability for programs, which means garbage collectors have more garbage to remove. Thus, stop-the-world pauses experienced by applications are increasing.

In some environments, application pauses are unacceptable; such a scenario is described within the section on real-time GC in [5]. In order to decrease or remove pauses, we want to use multiple processors and cores to share the work of both running an application and its GC. This is known as *parallel processing*, and it can be used to enhance performance. With demand for faster responding applications coupled with more memory needing managing, it is paramount that GC be optimized using parallel processing.

In Section 2, we cover background information on GC, parallel processing, and GC with parallel processing. From there, we focus our examination of parallel processing in GC. Section 3 covers the Continuously Concurrent Compacting Collector (C4), Section 4 considers the Collie collector, and Section 5 looks at a Field Pinning Protocol (FPP). Discussion of each is focused on how *compaction*, moving live objects to adjacent memory locations, in GC can run parallel to and independent of the application running.

2. BACKGROUND

2.1 Garbage Collection (GC)

Newly allocated objects in JVM languages are stored in a memory location called the *heap* [6]; this memory has no gaps, so it is contiguous. The heap represents the memory available to a program and is a virtual layer of memory that corresponds to physical memory. Thus, when GC removes garbage, it cleans up the heap. Applications access objects stored on the heap through *references*, fixed-size values that refer to virtual object locations in the heap [5]. An application uses a memory structure known as a *stack* as a workplace for methods being called, which is separate from the heap. Only references to objects, not the objects themselves, are stored on a stack. Objects can contain references to other objects, so references can also appear on the heap.

A full performance of an entire GC algorithm is known as a *cycle* and is usually started when the heap is full or nearly full. GC can be broken down into two major steps.

First, a GC algorithm performs *set condemnation* when deciding which objects are garbage. The garbage collectors we examine perform set condemnation using a method known as *tracing*. During this, objects are determined to be reachable by traversing references from global, root objects. Any objects reachable by chaining references from these objects could still be used by the application. Unreachable objects are garbage. Second, the algorithm performs *reclamation* by recovering the memory used by garbage objects.

To optimize GC, a variant can be performed known as *generational* GC. Generational collectors rely on most objects not living long on the heap [10]. Thus, GC efforts are focused on these objects. Typically, this is achieved by dividing the heap into two generations. One generation contains younger objects and is collected more frequently. If objects in the young generation of the heap survive enough GC cycles, they will move into the old generation. By not considering the entire heap with each GC cycle, an application experiences shorter disruptions.

As memory is reclaimed by a garbage collector, the heap is subject to *fragmentation* [10, 4, 8]. Fragmentation is the forming of interspersed locations of used space in contiguous memory. This is an issue as space for new objects being allocated on the heap becomes difficult to find and manage.

Consider an example with an object that uses 2 megabytes (Mbs) of memory, such as an array. Suppose that after several GC cycles, we have open memory spaces of at most 1 Mb. The JVM experiences additional overhead in allocating space for this object as it must be stored across non-contiguous locations. Additionally, more overhead occurs as accessing the object requires locating all of its parts.

Compaction fights fragmentation of the heap by moving live objects into a contiguous memory location, and it is performed as part of GC. Two steps are typically involved in compaction. After set condemnation, *relocation* moves live objects to a contiguous memory location. The contiguous memory location being moved to is often referred to as *to-space*. Likewise, an object is moved from *from-space*. While called relocation, objects are typically copied rather than moved. After relocation, a *remapping* phase updates all references to moved objects to refer to their new locations. When compaction is performed, reclamation is often done by marking from-spaces as freed.

2.2 Parallel Processing

To develop an understanding of parallel processing, we need to know what *processes* and *threads* are [7]. A process is an instance of a computer program being run. For example, a JVM and a word processor are two processes. A thread, at a basic level, is a component of a process; it performs an independent sequence of instructions from the process it belongs to. The order of instructions are preserved within a single thread, but not necessarily among multiple threads.

A process can involve a single thread or multiple threads. In the JVM, each thread has a personal stack to keep track of variables and references used within its specific set of tasks. Parallel processing is the utilization of multiple cores of processors to run threads concurrently. If processes are designed to run with multiple threads, the extra processing power can make the process more efficient.

Creating multi-threaded processes can result in many new issues. One of these issues, relevant to GC, is that modifications to an object can be lost. For example, consider a process with two threads, A and B. Suppose that thread A relocates objects in memory with copy operations, and thread B modifies properties of objects. A modification by thread B to an object could be lost if thread A relocates the object, but thread B modifies the object at the old location due to not knowing it was relocated. While the example is fairly simple, it can be seen how such a problem applies to the relocation phase of compaction within GC.

To deal with multiple threads relying on the same data, *synchronization* techniques are used. These can prevent issues like data corruption, described above, or even process-halting exceptions. One such synchronization technique is a *memory barrier* (referred to as a read barrier or simply a barrier), which is an instruction set that must be performed by a thread before accessing memory [11]. Uses of this include ensuring threads meet certain requirements before accessing memory or disallowing a thread to access memory while another thread is doing so. While barriers can help prevent conflicts, the checks performed result in extra processing overhead for the threads.

2.3 GC with Parallel Processing

We can now return to our original goal of applying parallel processing to GC. Threads that are part of the application process are referred to as *mutator* threads (or mutators) because they can mutate data [10, 4, 8]. Threads used by the garbage collector are simply GC threads.

A *parallel* garbage collector uses multiple threads simultaneously to perform GC. [9] All but one of the algorithms that we examine are parallel. A *concurrent* garbage collector executes instructions in parallel to the application running. In other words, the garbage collector does not entirely stop the world. A GC algorithm can be only partially concurrent or parallel. For example, a collector could only have the tracing phase be parallel while reclamation is concurrent, or a collector could be both entirely parallel and concurrent.

This paper focuses on concurrent compaction within GC. We do so due to the importance of avoiding stop-the-world pauses caused by garbage collectors. Additionally, compacting collectors are vital for efficient application performance.

One important metric for considering how a collector is affecting an application is *latency* [5]. Within GC, latency is a measure of how much a collector negatively impacts application processing performance. A higher latency implies the collector uses more processing resources that the application could be using. It can be measured in various ways but is consistently measured across collectors during a set of tests. As an example, one way to examine latency is viewing how server response times are slowed during GC.

3. THE C4 COLLECTOR

The first concurrent compaction algorithm we examine is implemented in the Continuously Concurrent Compacting Collector (C4), which is a garbage collector included in JVMs commercially shipped by Azul Systems [10]. The researchers, Iyengar et al., describe C4 as an enhanced, generational variant of the Pauseless GC algorithm [2]. C4 is generational in that the entire GC algorithm is independently utilized in both the young and old generations; in addition, both generations are concurrent with the application. C4's intended environment consists of large servers with multiple gigabyte heaps.
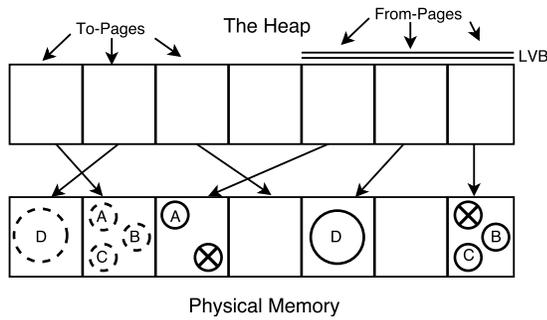
Figure 1: In C4, memory is relocated page-by-page. Dead objects are represented with X. From-pages are protected from mutators by the LVB.

3.1 Loaded Value Barrier

C4 uses a read barrier called the *Loaded Value Barrier* (LVB) to synchronize concurrent threads throughout the GC process [10]. The LVB places *invariants*, properties to be maintained, on each reference as it is loaded from memory. One of the invariants relates to the tracing portion of the GC algorithm, so it is not discussed here. The other invariant ensures that references loaded by a mutator during compaction point to safely accessible objects: objects that have already been moved. If the invariant does not hold when a reference is loaded, the barrier will trigger and correct the situation as we discuss in the next subsections.

3.2 Concurrent Relocation

The relocation phase of compaction in C4 occurs on a per-*page* basis, where a page is a fixed-length, contiguous block of virtual memory in the heap backed by a contiguous block of physical storage [10]. In such a model, from-space consists of *from-pages* and to-space of *to-pages*. To quickly empty pages, the most sparsely populated ones are relocated first, which also moves objects in physical memory. In Figure 1, the left-most from-page's live contents are relocated to a to-page first as the from-page has only one small object alive.

From-pages are protected by the LVB as can be seen in Figure 1. LVB will trigger for mutator threads encountering a reference to an object on a protected page. The mutator's subsequent behavior is:

- If the object is relocated already, find its new location

- If the object is being relocated currently, wait until the GC thread is finished

- If the object has not been relocated, move the object

In all cases, the mutator cannot use the object until it is on a to-page. Additionally, LVB has mutators correct the references to avoid further barrier triggers on the same reference. Without mutator interference, GC threads will simply relocate the objects but not remap references at this time.

When all live objects are relocated from a from-page, the C4 compactor will perform *quick release*. The objects have already been moved to to-pages, as shown by dashed circles in Figure 1, so the contents of the from-pages are no longer needed. Thus, the physical memory backing the page can be freed. The from-page will be in use as it still has references to it, but the physical memory can be recycled efficiently. In
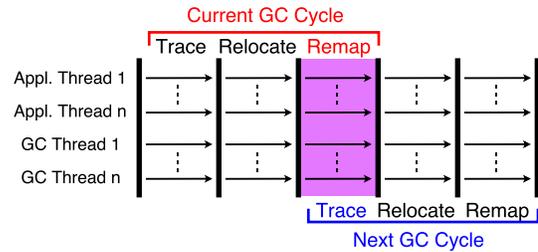

Figure 2: C4 GC cycle. Remapping and tracing are rolled into one traversal. (adapted from [10])

terms of Figure 1, the heap memory on top is used until after the remapping phase, but the physical memory below can be paired with a different virtual page and used immediately.

3.3 Concurrent Remapping

To maintain concurrency while updating all references to the now relocated objects, the C4 compaction algorithm uses two techniques [10]. The first is *lazy remapping*. With this, mutator threads continue updating references as they trigger the LVB. In order for the remapping phase to end, all live references must be updated. However, remapping could go on indefinitely if lazy remapping alone is performed.

To finish remapping, a traversal of live references must be performed to ensure all are updated; this is like the tracing traversal used to determine which objects are garbage as discussed in Section 2.1. No physical resources are being held due to quick release and lazy remapping does not severely impact mutator operations. As a result, the remapping traversal can wait until another GC cycle starts. C4 will perform the remapping of one GC cycle during the tracing of the next cycle since both tasks traverse the same references in memory. To visualize how this works, examine Figure 2. Upon remapping completion, all references to from-pages are now gone. Thus, the virtual addresses are freed and reclamation of memory is now complete.

3.4 C4 Results Summary

Experiments done with C4 evaluated improvements of using an algorithm that is simultaneously generational and concurrent [10]. C4 is tested against a non-generational C4 algorithm as well as two algorithms that perform non-concurrent compaction. All garbage collectors were tested on the same hardware; for specifications, see [10]. The test exhibited live sets of objects on the heap consistently at a size of around 2.5 gigabytes (Gbs); the actual heap size was allowed to grow, indicating greater amounts of garbage on the heap. The applications were run long enough to ensure multiple full-heap GC cycles ran and at least one significant compaction event occurred.

The primary performance metric monitored was worst-case response times of servers while experiencing the heap sizes described, which provides a useful indicator of latency. C4 maintained the smallest worst-case response times across the largest range of heap sizes. The worst-case response times were usually in the range of 0.01-0.1 secs; an application user would not notice such short pauses. The non-generational version of C4 could also reach low worst-case response times, but for a more limited range of heaps sizes. Standard C4 sustained the times for heap sizes of 5-35 Gbs;

however, the modified C4 could only do so for heaps sizes of 15-35 Gbs. The non-concurrently compacting collectors had multi-second worst-case response times for all heap sizes, which would cause noticeable pauses for a user.

4. THE COLLIE COLLECTOR

We now look at the concurrent compaction technique used by the single-threaded Collie garbage collector described by Iynegar et al. [4]. This is the same research group that worked on C4, so Collie utilizes modified versions of several techniques from C4, such as: the tracing algorithm, the LVB, page-granularity compaction, and quick-release behavior. Only the LVB is modified in significant ways as discussed below. Like C4, the Collie collector is designed with server environments and large heaps in mind. However, it is not commercially available. Collie is implemented and tested in the same production-quality Azul JVM the Pauseless GC algorithm is implemented in [2].

4.1 Transactional Memory

The Collie's compaction algorithm sets itself apart due to its use of a *transactional memory* (TM) system as a concurrency control alternative to barriers [4]. TM systems allow sections of code to function analogously to *transactions* from database systems, which are a series of operations performed as one unit where they occur in an all-or-none manner.

For example, consider updating sale prices of items in a store; this requires ensuring the store price matches the sale price advertised. When updating the prices for an item, both the advertised price and the store price should be updated. Only updating the advertised price could lead to irate customers, and only updating the store price could lead to customers not knowing about the sale. A transaction ensures an item's prices are unchanged if the update process is started and then aborted before completion. Additionally, customers cannot see the prices mid-transaction; they only see the price from before the transaction began.

TM systems allow a series of operations modifying memory to be placed in a *transactional procedure*, much like a database transaction. The TM system monitors access of concurrent threads to *transactional variables*, variables modified by a transactional procedure. Concurrent threads will operate in parallel until they try to modify the same transactional variable. How the conflict should resolve can be specified, but the general behavior would be for one transaction to be aborted and the other allowed to continue. When a transaction is completed, its changes are *committed*; any changes made to memory by the transaction are finalized [12].

Each object on the heap has a *referrer set* [4]. This is the precise set of all references pointing to the object. Referrer sets are stored as transactional variables since they are used by a transactional procedure (simply referred to as a transaction) that relocates and remaps a single object. An object must maintain a *stable* referrer set throughout the GC process. That is, its referrer set cannot be modified or expanded, which ensures that all of the object's references are correctly remapped.

4.2 The Collie Protocol

After a tracing phase at the start of GC, heap compaction happens during a *transplantation phase* [4]; transplantation is a term meaning the combination of relocation and remapping. Live objects and their referrer sets were determined
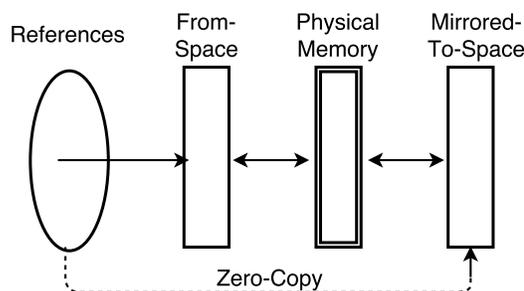

Figure 3: Mirrored-to-space and zero-copy transplantation in the Collie collector.

during tracing. As mentioned, the operations for object transplantation are stored as a transaction. Individual object transplantations are done only on objects that maintain stable referrer sets throughout the process. An object is deemed *non-individually transplantable* (NIT) if its transaction is aborted or it fails to maintain a stable referrer set.

Before performing the transactions to move individual objects, the objects need to be checked for having stable referrer sets. To do this, threads' stacks are checked for references to objects. An object with references to it from the stack of any active mutator thread will be marked NIT since the object is in use, which could cause referrer set modifications. This ensures GC does not interfere with mutator work. The LVB-style read barrier receives its first new purpose here. When a thread attempts to load a reference, and add it to its stack, the object the reference points to will be marked NIT as a mutator is using it.

The copying of an object actually occurs outside the transaction since any mutator access of the object will trigger the LVB-style barrier, rendering the object NIT. Thus, only remapping by updating the referrer set is done in the transaction, which looks like:

1. Transaction is started

2. Object is checked to not have been marked NIT

3. All references are changed to point to the object's to-space location

4. Memory transaction is committed

If a transaction fails or is interrupted, as detected by the TM system, the object is rendered NIT.

4.3 Aborting Compactor

After all individually transplantable objects are transplanted, we are left with unmoved NIT objects. [4]. However, compaction cannot terminate with objects in from-space. The additions of a *mirrored-to-space* and *zero-copy transplantation* are necessary to terminate compaction without disrupting mutators.

Mirrored-to-space is a virtual memory space corresponding to, and the same size as, from-space. It maps to the same physical memory as the corresponding from-space, as can be seen in Figure 3. For compaction termination purposes, mirrored-to-space is logically considered a part of to-space.

Zero-copy transplantation gets the compactor to believe it has finished a cycle. Mirrored-to-space holds the same objects in physical memory as from-space. Thus, if references are remapped to mirrored-to-space from from-space, the objects appear to be relocated without physically moving. This technique is zero-copy transplantation. In Figure 3, all references are to from-space for NIT objects. After zero-copy transplantation, they point to mirrored-to-space.

Zero-copy transplantation is performed on all NIT objects. The LVB-style barrier receives a second purpose here. As mutators access references to NIT objects before they are updated to mirrored-to-space, the barrier will cause the mutators to correct the references before using the object. Also like C4, entirely finishing zero-copy transplantation, which is just remapping, is rolled up into the next GC cycle's tracing phase. In the end, NIT objects are not actually compacted; for this reason, Collie is said to have an *aborting* compactor. It will cease moving objects to allow mutator threads to continue working uninterrupted.

4.4 Collie Results Summary

The main goal of the Collie collector is to decrease latency during compaction [4]. Collie was tested against a modified variant of the Pauseless GC algorithm [2]. Accordingly, both collectors were implemented within the same JVM. For exact testing specifications, refer to [4]. Collie utilizes several aspects of C4, which in turn builds upon the Pauseless collector. This provides an effective comparison to see how use of transactional memory and an aborting compactor affect application performance.

To examine latency, *minimum mutator utilization* (MMU) is measured. MMU is the minimum percentage of mutators being utilized over a period of time. Ideally, MMU would always be at 100% so all mutators are being used. MMUs were taken from various time periods during which compaction was running and was compared between the two algorithms. Comparisons were done with a 128 megabyte (Mb) heap and a 512 Mb heap. For all time windows, the Collie compactor had higher MMUs than Pauseless. Collie did not go below a 70% mutator utilization level whereas Pauseless averaged between 40% and 70%. This shows that the aborting nature of Collie lends itself to ensuring mutator response time, and thus decreasing latency, despite potentially limiting the amount of memory compacted.

5. FIELD PINNING PROTOCOL

Last, we examine concurrent compaction in GC through a Field Pinning Protocol (FPP), designed by Österlund and Löwe [8]. FPP describes a barrier-free process for performing the relocation aspect of concurrent compaction. The remapping phase and other portions of GC are left up to the host garbage collector. FPP is compliant with the Java Native Interface to allow for integration into a JVM. For purposes of testing, FPP is implemented within the Garbage-First GC algorithm in the HotSpot JVM of OpenJDK 9 [3], which lacks concurrent relocation.

5.1 Hazard Pointers

The key component driving FPP is the *hazard pointer* (HP) [8]. In FPP, all mutator threads contain a list of HPs. The HPs point to objects in memory a thread is accessing, or *pin* them. When a thread finishes using an object, it drops the HP. So long as an object is pinned by a mutator, it can-

not be relocated as it is still in use. Once all threads have unpinned the object, it can safely be relocated. When discussing the theoretical construction of the algorithm, a granularity of pinning individual fields is used, thus the name *Field* Pinning Protocol.

While it is explained in more detail below, an example of how HPs work on a high-level is useful. Imagine a pot of coffee at work. Anyone wanting coffee must make it known to others by having a cup, otherwise the coffee is liable to be moved to another break room. The people are effectively pinning the coffee. Once everyone has had their share of caffeine, it should be moved. This can be done safely when no one has cups any longer, so the coffee is not pinned.

5.2 Collaborative Copying and Blame

HPs prevent premature relocation of objects by serving as a distributed counter. Threads cannot relocate objects with a non-zero HP count. A mutator thread *impedes* a copying thread if its HP prevents the copying. The impeding thread is *blamed* for the interruption [8]; it becomes responsible for ensuring the object is relocated. GC threads cannot receive blame as they will not impede copy attempts.

The process followed by mutator threads during the relocation phase when accessing an object is as follows:

1. Pin the object by adding a HP

2. Determine whether the object has already been copied

3. Mutate/use the object. Since the object is pinned, the thread can do so without worry

4. Unpin the object when no longer needed

5. If the thread was blamed due to impeding another thread's copy attempt, attempt to copy the object:

   (a) Check other threads for HPs before copying. If HPs found, blame all the threads and move along

   (b) Proceed to copy the object with new pins to the object impeding the copy attempt

Mutators will continually bounce blame around until no threads have the object pinned. At this time, the last thread to attempt to copy the object will succeed.

Reconsider the coffee example. Suppose someone is in charge of moving the coffee to the other room, much like how a GC thread must relocate objects. They can try, but others could still have cups and want more coffee. The interrupted person has other tasks they could be doing, so they tell everyone with a cup to move the coffee when finished drinking it; they blame the others for not finishing the task and make them responsible for it. When those blamed finish their coffee, they will try to move it as well. If interrupted by others with coffee cups, HPs, they pass the blame along. This continues until a last person finishes their coffee and moves it to another room.

At the start of FPP relocation, GC threads will attempt to copy live objects. At this time, blaming is disabled. After a first round of attempting to copy objects, GC threads will again try to copy the remaining objects. Any mutator threads impeding copy attempts at this time will be blamed. Blame continues to be passed until the objects are relocated.

It is possible for an object not to move because of being continually pinned. How this terminates depends on

the host GC algorithm used. When using the Garbage-First (G1) algorithm, this process goes until all objects are moved or another GC cycle starts. If objects have not relocated when another cycle begins, they are automatically marked for relocation again. Allowing mutators to continue passing blame works in G1 because, like C4 and Collie, the algorithm will roll up remapping into the next GC cycle's tracing phase. Possible alternative approaches include setting a max number of impediments or setting a timer; after such a threshold is reached, the mutator threads pinning the object will be stopped and the object will be relocated.

5.3 FPP Results Summary

As mentioned, FPP was implemented in the G1 garbage collector for testing purposes [8]. Appropriately, the modified G1 collector that uses FPP for relocation is tested against the unmodified G1 collector. Tests were run in a MacBook Pro; for the hardware details, see [8]. We focus on the tests comparing latency of the two garbage collectors. For other tests performed and their results, see [8].

For examining latency, the h2 benchmark from DaCapo is used [1]. The benchmark was chosen due to being memory intensive, which makes latency issues apparent. Latency is measured using the jHiccup tool developed by Azul Systems. Compared to its host garbage collector, G1 with FPP improved latency for all time intervals examined. On average, latency was improved by around 50%; application pause times were about half as long. Latency was shown to mostly come from host garbage collector activities such as tracing and remapping rather than delay from dealing with pins.

6. CONCLUSIONS

We have examined three techniques for concurrent compaction that all effectively reduce latency of GC. The full GC algorithms were tested for latency in distinct environments, with different benchmarks, and against various algorithms, so it is difficult to compare them to one another. Additionally, direct comparisons are especially difficult since the algorithms were not all designed for the same environment.

The algorithms we have looked at achieved concurrency while maintaining low latency in various ways. C4 and Collie use read barriers differently. C4 uses a barrier to control mutator behavior, and Collie uses one to allow mutators to work without interruption. FPP takes a largely different approach by focusing on performing solely relocation without barriers; it avoids standard synchronization methods by using HPs and thread collaboration instead. Overall, the transition from barrier to mostly barrier-free concurrent compaction shows advancements in the field. Regardless of approach, concurrent compaction is necessary for efficient GC with growing heaps and increasing environmental demands. Ultimately, the approach for compaction used will depend on its intended environment.

Acknowledgments

7. REFERENCES

[1] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, 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 *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[2] C. Click, G. Tene, and M. Wolf. The Pauseless GC algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 46–56, New York, NY, USA, 2005. ACM.

[3] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-First garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM.

[4] B. Iyengar, G. Tene, M. Wolf, and E. Gehringer. The Collie: A wait-free compacting collector. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 85–96, New York, NY, USA, 2012. ACM.

[5] J. Lindblom. Modern considerations of garbage collection in the Java HotSpot Virtual Machine. In *UMM CSci Senior Seminar Conference*, VEE '05, Morris, MN, USA, 2011. UMM.

[6] Oracle. Understanding memory management, 2014. `docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html`, [Online; accessed 24-October-2015].

[7] Oracle. Processes and threads, 2015. `docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html`, [Online; accessed 24-October-2015].

[8] E. Österlund and W. Löwe. Concurrent compaction using a Field Pinning Protocol. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2015, pages 56–69, New York, NY, USA, 2015. ACM.

[9] W. Puffitsch. Hard real-time garbage collection for a Java chip multi-processor. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 64–73, New York, NY, USA, 2011. ACM.

[10] G. Tene, B. Iyengar, and M. Wolf. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 79–88, New York, NY, USA, 2011. ACM.

[11] Wikipedia. Memory barrier — Wikipedia, The Free Encyclopedia, 2015. `en.wikipedia.org/w/index.php?title=Memory_barrier&oldid=663234890`, [Online; accessed 24-October-2015].

[12] Wikipedia. Transactional memory — Wikipedia, The Free Encyclopedia, 2015. `en.wikipedia.org/w/index.php?title=Transactional_memory&oldid=680734631`, [Online; accessed 3-October-2015].