ABSTRACT

IYENGAR, BALAJI. Non-Blocking Algorithms for Garbage Collected Environments. (Under the direction of Edward Gehringer.)

Garbage collection has been one of the primary reasons for the increasing popularity of runtime based languages such as Java™. A bulk of the current enterprise workloads run on the Java™ runtime platform. However, garbage collection algorithms that are currently shipping as part of Java™ runtime environments for the past decade are starting to be the source of bottleneck in scaling application throughput as well as in meeting application response time requirements. This is primarily an artefact of the ever increasing application memory footprints and the inability of current garbage collection algorithms to deal with this without affecting application performance.

The large and variable cost of synchronization between the application and concurrent compacting garbage collectors is the primary reason for jitters in application responses and throughput. We describe two different concurrent compacting collectors that aim to reduce this cost. We introduce new terms and requirements useful for analyzing concurrent compacting collectors, including definitions of referrer sets, object transplantation and the distinction between individually transplantable and non-individually transplantable objects. Our collectors build on these terms and the notions of an object's legal transplantation states to support concurrent compaction.

The Collie collector, the first of these collectors, uses a combination of read barriers and hardware transactional memory operations to support concurrent compaction. The collector uses a wait-free, constant time read-barrier that supports direct object references and does not pay the constant indirection cost of Brook's [19] style barriers or handle based heaps. It is demonstrated using the multi-address atomicity feature, a form of hardware transactional memory supported by the Azul Vega architecture. The Collie collector is the first concurrent compacting collector based on a transactional memory implementation.

The second concurrent compacting collector described int this dissertation, i.e., the single-stateword collector, also implements a wait-free read-barrier without needing the multi-address atomicity semantics provided by hardware transactional memory. Both these collectors are able to provide significant MMU improvements even in the sub-5ms time windows compared to the Pauseless collector [21], an industrial strength concurrent compacting collector, on the Vega architecture and match it in its ability to scale to large heap sizes and provide high throughputs.

The second part of this dissertation focusses on improving the scalability of the parallel marking phase of a garbage collection algorithm. Parallel marking algorithms use multiple threads to walk through the object heap graph and mark each reachable object as live. Parallel

marker threads mark an object "live" by atomically setting a bit in a mark-bitmap or a bit in the object header. Most of these parallel algorithms strive to improve the marking throughput by using work-stealing algorithms for load-balancing and to ensure that all participating threads are kept busy. A purely "processor-centric" [42] load-balancing approach in conjunction with a need to atomically set the mark bit, results in significant contention during parallel marking. This limits the scalability and throughput of parallel marking algorithms.

We describe a new non-blocking and lock-free, work-sharing algorithm that aims to reduce contention during atomic updates of the mark-bitmap by parallel task-threads. Our work-sharing mechanism uses the address of a word in the mark-bitmap as the key to stripe work among parallel task-threads, with only a subset of task-threads working on each stripe. This filters out most of the contention during parallel marking with ~20% improvements in performance. We prototype the algorithm within the C4 [63] collector that ships as part of an industrial strength JVM for the Linux-X86 platform.

Non-Blocking Algorithms for Garbage Collected Environments

by
Balaji Iyengar

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2012

APPROVED BY:

Yan Solihin	Matthias Stallmann
Huiyang Zhou	Edward Gehringer Chair of Advisory Committee

DEDICATION

To my parents.

BIOGRAPHY

The author was born in Mumbai, India in 1980. He attended the University of Mumbai from 1997-2001, graduating with a bachelors degree in computer engineering in 2001. He then began his graduate studies in the Department of Electical and Computer Engineering at North Carolina State University in 2001. He completed his Masters in 2003 and started his Ph.D. program under the guidance of Dr. Edward Gehringer in 2004. After an internship at the Sun Microsystems Research Laboratory, he joined Azul Systems Inc. in 2005. Balaji has been part of the Java virtual machine team at Azul Systems Inc. for the past 6 plus years, helping ship an industrial strength JVM that includes an implementation of the state of the art, C4 garbage collection algorithm. He re-enrolled in the Ph.D. program in 2011.

His research interests span a wide array of topics in computer science. He is specifically interested in scalable, parallel algorithms, virtual machines and concurrent garbage collection algorithms.

ACKNOWLEDGEMENTS

I would like to begin by expressing my sincere thanks to my research advisor, Dr. Edward Gehringer. He took me under his tutelage in 2004 as a doctoral candidate and has been an exemplary source of both academic and non-academic guidance. He has invested hours, sculpting my abilities as a researcher, providing valuable feedback and improving my abilities to express my ideas in a clear and technically rich manner. A significant portion of my doctoral program was spent with me being remote and Dr. Gehringer's dedication and persistence were of immense help in navigating the special set of challenges introduced by the geographical separation. In addition, I have thoroughly enjoyed all his classes. I sincerely appreciate his effort and dedication towards my future and the success of my research.

I am also extremely appreciative of the members of my advisory committee, Dr.Yan Solihin, Dr.Matthias Stallmann and Dr.Huiyang Zhou. They have provided me with valuable and timely feedback and guidance. They were very supportive during my remotely conducted oral preliminary examination. I sincerely thank them for the time and energy they have invested in my doctoral program.

I have spent the last 6 plus years working on various aspects of the Java™ virtual machine developed by Azul Systems. Azul Systems has built a custom system (CPU, chip, board, and OS) specifically to run garbage collected virtual machines. The custom CPU includes a number of features that have allowed Azul to design, implement and ship an industry leading concurrent and parallel garbage collector. The custom CPU includes the only currently shipping implementation of hardware transactional memory. This amazing piece of technology served as the platform on which I have developed and implemented my research algorithms. This research effort would have been a significant challenge to implement on any other platform. I truly believe that having access to this platform allowed me to fundamentally break new ground in garbage collection research and in the use of new concurrency primitives such as transactional memory. These new primitives are now starting to appear on standard commodity platforms. I am truly indebted to everybody who had a hand in the design and implementation of this engineering marvel. I would like to express my deepest gratitude and thanks to the founders of Azul, the hardware design engineers, the kernel engineers, the JVM engineers and last but not the least the QA engineers.

My experience at Azul has exposed me to an unparallelled level of technical expertise, technological challenges and most of all the amazing professionalism of my colleagues. Azul to me, has been an extended graduate school and a richer learning environment would be hard to find. I would like to express special and sincere thanks to Michael Wolf and Gil Tene, my colleagues at Azul, who have spent hours discussing technical ideas with me. Michael has been the best mentor one could hope for. His breadth and depth of computer science knowledge is inspritational as is his ability to lay out a complex technical argument in the most lucid manner. Gil's ability to dissect and present a technical discussion is nothing short of an art form and I sincerely hope that some of it has rubbed off on me. Gil spent an immense amount of time helping me iron out the subtle details and the inevitable race conditions while I was designing and writing up the algorithm that forms the core of this dissertation. I would also like to thank my manager at Azul, Bean Anderson, who has been extremely supportive of my endevor to finish my doctoral research.

On a non-technical note, Gil and Joe Coha of Azul Systems, helped my wife and I, plan our first backpacking trip in the Sierra's and several other similar hikes. I have also had a lot of fun playing foose-ball and ping-pong with Laksh, my colleague and cube-neighbour. All these have helped me immensely in maintaining my sanity over the past year and a half, while I worked full time, took courses and worked on my research.

To my parents, I can't ever fully thank them for their dedication towards my success and their tireless efforts to expose me to the best of opportunities. I thank them for all the sacrifices they have made over the years to provide me with everything I have ever needed. My elder sister and her husband, have been a constant source of support, guidance and inspiration over the years. She has supported me in every possible way over the years and I truly feel gifted to have her as my elder sister. My niece and nephew have always helped me keep things in perspective and their wishes during this journey have been a constant motivational source.

To my wife, Deepa, this would have been impossible without her support and encouragement over the last year and a half. She has endured my long work hours in silence and has helped in every possible way. Her love and support have been of tremendous help through all the tough times during this process. Lastly, but certainly not the least, I would like to acknowledge our amazing dog, Rhea. She is the smartest and the most loving dog one could have. She brings immense joy to us on a daily basis and the "Collie" algorithm, named after her breed, truly signifies how much she means to me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The performance of microprocessors has been driven my Moore's law for quite some time now. Memory chips have simply become faster over time on a course described by Moore's law, while processor speeds have often increased at a rate faster than that predicted by Moore's law. Advanced micro-architecture techniques that exploit parallelism in the instruction stream, i.e., *instruction-level parallelism*, combined with deep processor pipelines and sophisticated branch prediction techniques, are primarily responsible for faster growth in processor speeds. On-chip caches have helped bridge the gap between processor and memory speedups. All these have resulted in system architectures being able to extract *parallelism* from software in a manner transparent to the programmer, keeping their tasks simpler. Programmers have benefited from the resulting software speedups.

Unfortunately, it is becoming increasingly difficult for processor designers to further enhance the speed of modern processors by using these techniques. Typical instruction streams have only a limited amount of usable parallelism among instructions. Processors that issue more than 4 instructions in a cycle achieve very little incremental speedup, while paying high costs in terms of power and complexity. *Power* has become the most limiting factor for modern processor design. Modern processors require up to 100W of power, necessitating advanced and expensive cooling techniques. Given this "power ceiling," processor architectures must limit the amount of wasteful speculation and use other approaches to further reduce execution times.

In the past decade or so, in response to limiting factors such as *power* and the limited amount of easily exploitable instruction-level parallelism processor designs have moved away from deeper faster pipelines in uniprocessor systems. The paradigm shift has been towards chip-multiprocessors, or CMP, where multiple cores are integrated onto a single chip. Each core has a shallow pipeline, operates at lower frequencies and have very little speculative mechanisms built in. This paradigm is also commonly referred to as *throughput computing*. This architecture maps well to the client-server model, where an individual server has to process several hundred concurrent client requests. Individual requests have “soft” latency bounds, and sustaining throughput is critical, i.e., there should be no dropped requests. Throughput computing is thus able to support a larger number of concurrent requests at lower power.

Meanwhile, software has undergone a transformation to keep up with the paradigm shift in hardware design towards multi-core architectures. This was necessary to prevent software from becoming the bottleneck. Multi-threaded programming has become more universal. Applications built using modern programming languages, such as Java™ [34], that run on top of managed runtime environments, i.e., virtual machines, have built-in support for “threads”, as well as support for synchronization primitives such as locks. The libraries that ship with Java™, i.e., the JDK, also ship with thread-safe implementations of commonly used data structures such as hashtables [41]. These features, as well as automated memory management, i.e., *garbage collection*, have resulted in wide spread adoption of Java™ in the enterprise markets. Currently, by some estimates Java™ accounts for about 80% of the enterprise software market.

Garbage collection has been an ymportant aspect of the Java™ language. It has been critical to increasing programmer productivity and the robustness of programs. Without garbage collection, programmers, have to write their own memory-management modules. These tend to spring memory leaks and cause memory corruption, especially as the program scales to use more memory. This is not a reflection on a programmer's ability but a direct reflection of the challenges involved in implementing a scalable and thread-safe memory management module. Garbage collection significantly reduces memory leaks and memory corruption, thereby increasing program robustness, and allowing programmers to focus on implementing the core business logic of their application. Automated memory management has gotten so much traction in recent years that even older languages such as C++ are redefining their specification to include garbage collection. Newer languages such as Scala [46], tend to be runtime-based languages that support garbage collection.

This dissertation focuses on the effect of garbage collection[1] on application response times and throughput, as well as its effectiveness in performing its primary functionality, i.e., recycling memory. We begin with a short overview of garbage collection in section 1.1. The next two chapters of this dissertation focus on reducing the cost of GC-induced synchronization and making it predictable. We set the stage with a discussion of the need for synchronization between the garbage collector and the application threads, the synchronization mechanisms used for this purpose and their impact on the application performance in section 1.2.

The concurrent garbage collection algorithm, called the Collie collector(Chapter 2), makes use of hardware transactional memory, a new concurrency primitive used for multi-address atomic updates. Transactional memory(TM) has been an active area of research for the past couple decades; we give an overview of the design space and discuss the some of the general

[1] We will refer to garbage collection as GC or simply as the collector.

techniques used in the implementation of hardware transactional memory in section 1.3.

The final aspect of this dissertation deals with improving the performance of *marking*, a core functionality shared by several garbage collection algorithms. We briefly discuss the design variables in implementing a high-performance, concurrent marking algorithm in section 1.4.

We build on these to highlight the research problems that this dissertation deals with and to layout the structure of the dissertation in section 1.5.

1.1 Garbage Collection Overview

The garbage collector's first task is to identify parts of the memory that are accessible to the application program. Java being an object oriented program, the garbage collector views memory as a collection of objects. Each application program thread has its own execution context. This generally includes the thread stack, the registers and certain special pointers to objects, referred to as "handles," that are stored in special thread-local areas. The execution context of an application thread can hold pointers to objects in memory. The objects in memory, accessible via the pointers in the execution contexts of application threads, are considered reachable or live and are not recyclable by the garbage collector. In addition, objects that can be reached by following pointers from a live object are also considered to be live. The objects that are directly reachable via pointers in the execution context of the application threads, are commonly referred to as the "root-set." The garbage collector walks the entire heap graph starting from the root-set, marking every object it reaches as live. Marking an object generally implies setting a bit associated with the object address that identifies the object as live. This phase is commonly referred to as the *marking phase*.

Once the garbage collector has finished scanning the heap and has identified a set of live objects, its next task is to free up the unused memory. There are a couple different ways that a garbage collector recycles unused memory. The first method is to "sweep" the garbage objects, sort them by size and return them to free lists of memory blocks. These freed memory blocks are then used to satisfy allocation requests by application threads. By definition, application threads can't access garbage objects; hence the garbage collector can recycle these objects without a need for additional synchronization with the application threads.

Reclaiming garbage via sweeping, while being straightforward, has its drawbacks. Allocation requests that cannot use up an entire free block of memory and leave very little to satisfy the next request, cause memory wastage. This is referred to as internal fragmentation and free lists of memory blocks are prone to it. Sweeping by itself can't deal with fragmentation. The other drawback is the slowdown caused on the allocation path. An allocation request by an application thread has to find a block of the right size in the free list and pull this block off the list. An application thread can use bump-pointer allocation once it has a free block. Free-list management on the allocation path can slow down the supported allocation rate, thereby adversely effecting application throughput. Application throughput is directly related to how fast an application can allocate, and therefore free lists are not the most desirable allocation mechanism.

The other approach to recycle unused memory is to defragment the heap by copying live objects into new contiguous locations and freeing up memory where they are currently located. Objects are packed next to each other while being copied, avoiding GC-induced fragmentation. This also results in large chunks of memory being freed up. These larger chunks of memory are now handed out to application threads as thread-local allocation buffers to satisfy allocation requests. Application threads can use a fairly straightforward bump-pointer allocation [67] within these thread-local buffers. This significantly increases allocation speeds, which results in increased application throughput.

Garbage collection algorithms generally operate in two distinct phases, i.e., a mark-phase and a sweep-phase or a compact-phase. A separate family of collectors fuses the mark and the compact phases into a single “copying” phase. Hence, these are classified as “copying” collectors. This dissertation focuses on the “mark-compact” family of collectors, since they have distinct advantages over the other family of collectors [42] and find predominant use in modern JVMs.

1.2 Synchronization between the Garbage Collector and Application Threads

We will now discuss the need for synchronization between concurrently executing application threads (*mutators*) and the collector. The execution context of an application thread is mutable, i.e., is constantly changing as the program continues its execution. The garbage collector’s view of the application thread’s execution context can differ from that of the application thread’s view of its own execution context. Thus there is a need for synchronization between the collector and the application threads, i.e., mutators. As described previously, a mark algorithm walks the object heap graph and marks the reachable objects as live, while the application threads continue to run and mutate the heap graph. Starting from the root-set, for each object that a collector reaches, it does the following:

1. marks it live.
2. scans the “outwardly-directed” pointers in the object and proceeds to walk through them in a breadth- or depth-first manner.

A heap object is considered to be fully processed by the collector once it has performed both the abovementioned steps for that object. A collector will not revisit a heap object it considers processed. In the absence of synchronization between a mutator and the collector, a mutator thread can write a pointer to a yet unvisited object into an object that the garbage collector considers to be "completely" processed. This freshly written pointer will not be seen by the garbage collector since it considers the object the pointer is written into to have already been visited or processed. This can result in heap corruption if this pointer turns out to be the only pointer to the object; in this case the collector can't reach the object via any other path and it ends up recycling it as garbage.

This is illustrated further in figure 1.1. The figure shows a timeline of the object graph in the heap. The figure illustrates the snapshot of the object graph at four different times. In step 1, the garbage collector has already marked through object A, indicated by the black shade. The garbage collector will not revisit object A after this point, since it considers object A to be visited or completely processed. It has also marked object B as live, but is yet to traverse through the outward pointers emanating from object B. This state is indicated by the gray shaded object B. As can be seen, object A and B are directly reachable via the root-set. Object C is reachable via object B. Object C hasn't been visited by the garbage collector; this state is indicated by the white shade.

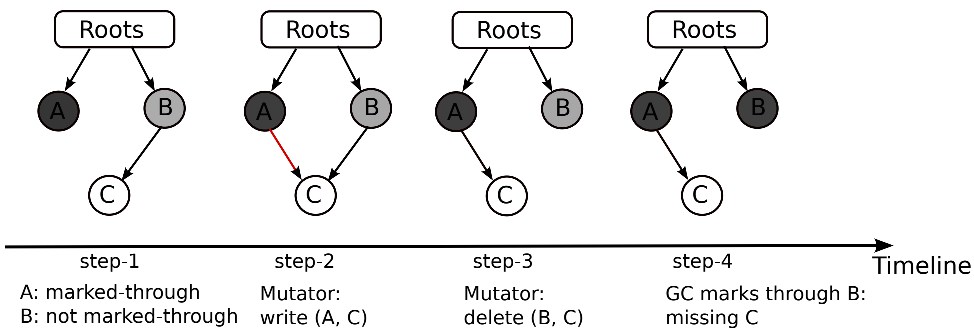

Figure 1.1: Heap Graph Timeline

In step 2, the mutator writes a new pointer from object A to object C, and in step 3, proceeds to delete the pointer from object B to object C. In step 4, the garbage collector starts to mark through object B. The garbage collector doesn't see object C, since it is not going to revisit object A and the pointer from object B to object C has been deleted by the mutator thread. Unless there is some sort of handshake between the concurrently executing, collector and mutator threads, the garbage collector won't be able to find the only pointer to object C. This is also referred to as the "lost-object problem" [42] and highlights the need for synchronization between the mutator and the collector thread during concurrent/incremental marking.

In addition to being able to track pointers in heap objects, the collector also needs to track pointers to objects in mutator execution contexts, i.e., the stacks and the registers. There needs to be a low-overhead synchronization mechanism that allows the collector to track down all the object pointers in the mutator stacks and registers. We discuss this further in section 1.2.1.

Concurrent heap compaction follows the mark phase in a garbage-collection cycle and involves copying/relocating live objects in discontiguous locations to contiguous memory locations. This results in a compact packing of these objects, allowing the garbage collector to recycle the original memory locations of the relocated objects.

Concurrent compaction has to meet a few requirements, listed below:

- Copy the object to its new location in a coherent manner, i.e., concurrent updates to the original object by the mutator should not be missed.
- Update references to the old location of the object to make them point to its new location.
- Prevent propagation of pointers to the old location of a relocated object.
- Prevent access to the old location of a relocated object once it has been copied to its new location.

In order to meet these requirements, the collector needs to synchronize with the concurrent executing mutators during the compaction phase. Concurrent marking poses a multiple-readers, single-writer coherence problem, since both the mutator and the collector read pointer fields but only the mutator can modify them. Concurrent compaction on the other hand poses a multiple-readers, multiple-writers coherence problem, since both the collector and the mutator are writing to the heap. Having discussed the need for synchronization between the collector and the mutators during both the phases, we will now discuss the various forms of synchronization and their uses in both the collector phases.

1.2.1 Stop-the-world Synchronization

Stop-the-world synchronization is a heavyweight form of synchronization between the collector and the mutator. It involves bringing all the application threads to a stop at the start of a GC phase and restarting them at the end of the GC phase. This precludes all races between the collector and the mutator. In addition, an entirely stop-the-world collection cycle allows the collector to identify a “precise” and correct live-set and relocate this live-set in a thread-safe manner.

Stop-the-world synchronization is implemented by suspending the mutator execution at certain “safe” points in their code. Application threads generally execute code emitted by the

JIT compilers; GC “safe” points are places in the application code where a mutator execution can be suspended without violating the legal language and platform specification. Most systems have short sequences of code that must be run in their entirety in order to preserve the invariants relied on by the collector. We cannot have a GC “safe” point in the midst of this section of code. An example is code executed by the mutator to setup an execution stack frame as well code for initializing newly allocated objects. GC safepoints also provide the collector with precise information on the location of object pointers in the stacks and registers. This is generally implemented using stack maps [42] that are bitmaps indicating the precise location of object pointers in the stack frames. The JIT compiler is generally tasked with the duty of generating stack maps. Most systems allow garbage collection at certain restricted safepoints, and produce stack maps only for those points.

The minimal set of safepoints needed for correctness includes each allocation, and each call of a routine in which there may be allocation or which may cause the thread to suspend in a wait. Beyond the minimal points needed for correctness, a system may wish to allow garbage collection at more locations so as to guarantee that garbage collection can proceed without an unbounded wait for the thread to reach its next safepoint. To make this stronger guarantee, there needs to be a safepoint in each loop; safepoints are generally placed at each backwards branch in a function. Function entries and returns can also be safepoints. These additional safepoints do not do anything that actually can trigger a garbage collection; hence they need to have an added check for whether garbage collection is needed/requested. Agesen [5] describes and compares the two ways of causing a thread to suspend at a GC-point. One is *polling*, where a thread checks a flag that indicates that a garbage collection has been requested. The other technique is *patching*, and involves modifying the code at the next GC-point(s) of the thread so that when the suspended thread is restarted, it will stop at the next GC-point.

Stop-the-world synchronization is not desirable, since it adversely effects the application throughput and response times. Application threads can’t respond to transaction requests during a stop-the-world pause, resulting in a visible jitter in the application behavior. The duration of an application pause caused by stop-the-world marking or compaction is directly proportional to the size of the live-set of the program. Enterprise applications that can consume 10s of gigabytes of heap memory can suffer from multi-second stop-the-world pauses induced by GC. These lengthy pauses are also unpredictable since they correlate with the size of the live-set, which varies with program phases. These drawbacks of a stop-the-world synchronization drive the need for a finer-grain synchronization between the collector and the mutator threads; we will now discuss these.

1.2.2 Barrier Synchronization

Memory-management systems augment individual application pointer[2] reads/writes with additional functionality that turns the operation into a *barrier*: an action that results in synchronous or asynchronous communication with the collector. These provide finer grain synchronization compared to the stop-the-world synchronization and find extensive use in all concurrent phases of the garbage collection. We will discuss the types of barriers and some general implementation techniques; the particular barrier code depends on the collection algorithm. In abstract terms, barrier mechanisms are used by the collector to *detect* and *act on* "interesting" pointers. The notion of what counts for an "interesting" pointer depends on the collection algorithm. *Detection* is the process of determining that a pointer is interesting. Acting on an interesting pointer generally involves recording the pointer for later use by the collector and in some cases might involve additional action such as "correcting" the pointer [63, 21].

Barriers are generally small sequences of code emitted by the JIT compiler prior to or following a pointer read/write. A typical barrier involves one or more checks that guard an action. Typical checks include null-checks and generation-checks on the pointer of interest. Most barrier implementations include a fast-path that is inlined for speed and a slow-path that the fast-path code can call if necessary. A well designed fast-path code can improve instruction cache performance and reduce the overhead on the application code.

Write barriers track pointer writes in application code and generally record the pointer of interest, which is either the pointer being overwritten or the new pointer value being stored. The flavor of the write-barrier depends on the collection algorithm and the intended use of the barrier. Write barriers also use a variety of data structures to record these pointers, such as card tables [59, 66], sequential store buffers [16] and hashtables [33]. While the common write-barrier implementations use inlined code sequences, the early flavors of write-barriers relied on operating system and hardware support. Tagged pointer architectures allowed pointers and non-pointers to be discriminated, and a hardware write barrier could set bits in a page table [50]. However, it is possible to use operating-system support to track writes without special-purpose hardware, For example, Shaw [56] modified the HP-UX operating system to use its paging system to record dirty pages. Boehm et al. [17] use write-protection on pages after a collection; the first write to a page since it was protected leads to a fault, and the trap handler sets the dirty bit for the page before unprotecting it to prevent further faults.

Read barriers track pointer reads in application code and are generally used by copying/compacting collectors to support concurrent relocation of objects[3]. In case of copying/compacting collectors, it is often the case that there are multiple copies of a single object during the process of relocation; these collectors use read-barriers to enforce an invariant that all mutators see a

[2]There are flavors of write-barriers that track all writes, i.e., both pointer as well as scalar writes.

common and current version of the object. Baker [13] read barrier was among the earliest to support incremental collection to enforce a to-space invariant. This was extended by Halstead [35] to support concurrent copying. A Baker style read barrier tracks a pointer to the from-space location of an object and forwards it to the corresponding to-space location of the object, copying the object in the process if necessary. The Brooks [19] read barrier is an indirection barrier where all heap accesses, of pointers, non-pointers and mutable values in header words, require an unconditional dereference operation to follow an indirection pointer. The indirection pointer points to the "current" location of the object. Brooks style read barriers are generally used in conjunction with write barriers to ensure termination of the GC phase [19].

Read-barriers, similar to write-barriers, can be implemented as inlined code [63] or by using virtual memory protection techniques. Appel et al. [7] and the Compressor algorithm [44] use similar virtual-memory page-protection primitives. Protection serves as the read barrier for concurrent copying collection, in order to prevent the mutator from accessing to-space pages whose objects have not yet been copied or contain unforwarded pointers. Ossia et al. [52] also use protection to allow concurrent forwarding of pointers in pages containing compacted objects.

Barrier implementations often use underlying hardware concurrency primitives such as the compare-and-swap (CAS) instruction [53, 63, 47]. Barriers that use page-protection techniques are generally expensive since they incur the penalty of switching to kernel code. They are also prone to "trap storms" as many protection faults are triggered immediately after a collection to unprotect the program's working set [44, 21].

Barrier implementations also differ in the kind of progress guarantee [39] they provide to the application code. Most barrier implementations provide lock-freedom and obstruction-freedom.

Considering that barriers are executed by the application code at every pointer read/write, application response times and throughput are stronly impacted by the concurrency primitives used in the barrier code, the design of the barrier-fast-path code and the progress-guarantee provided by the barrier code. The concurrent, collection algorithms proposed in this dissertation provide the first of its kind, constant-cost, "wait-free" [39] read-barrier.

1.3 Transactional Memory Overview

As mentioned earlier, our proposed concurrent compaction algorithm, the Collie collector, uses hardware transactional memory to support concurrent compaction. We round off this chapter with an overview of the transactional-memory design space and a discussion of techniques used in the implementation of hardware transactional memory.

Lomet [45] was the first to observe that an abstraction similar to a database transaction might make a good programming language mechanism to ensure the consistency of data shared

[3] In some cases, read-barriers are used to support concurrent marking [21, 63]

among several processes. In 1993, Herlihy and Moss [38] proposed a hardware-supported transactional memory, and Stone et al. [61] proposed an atomic multi-word operation known as "Oklahoma Update" to rekindle research in this area.

A transaction is a sequence of actions that appears indivisible and instantaneous to an outside observer and has four attributes (called the ACID properties):

1. *Failure atomicity*: transaction completes successfully or seems like it did not start.

2. *Consistency*: a transaction should leave the world in a consistent state.

3. *Isolation*: the result should be correct independent of other concurrent transactions.

4. *Durability*: the result should be permanent once the transaction commits.

The properties of transactions provide a convenient abstraction for coordinating concurrent reads and writes of shared data in a concurrent or parallel system. Today, this coordination is the responsibility of a programmer, who has only low-level mechanisms, such as locks, semaphores, mutexes, etc., to prevent two concurrent threads from interfering. Even modern languages such as Java and C# provide only a slightly higher-level construct, a monitor, to prevent concurrent access to an object's internal data. These low-level mechanisms are difficult to use correctly and are not composable.

Transactions provide an alternative approach to coordinating concurrent threads. A program can wrap a computation in a transaction. Failure atomicity ensures that the computation completes successfully and commits its result in its entirety, or else aborts. In addition, isolation ensures that the transaction produces the same result as it would if no other transactions were executing concurrently. Although isolation appears to be the primary guarantee of transactional memory, the other properties, namely failure atomicity and consistency, are important. If a programmer's goal is a correct program, then consistency is important since transactions may execute in unpredictable orders. It would be difficult to write correct code without the assumption that a transaction starts executing in a consistent state. Failure atomicity is a key part of ensuring consistency. If a transaction fails, it could leave data in an unpredictable and inconsistent state that would cause subsequent transactions to fail. Moreover, a mechanism used to implement failure atomicity, reverting data to an earlier state, turns out to be very important for implementing certain types of concurrency control.

1.3.1 Transactional Memory: Design Space

A transactional memory researcher has to deal with several design choices. We summarize some of them below:

- **Strong versus weak isolation**: Weak isolation guarantees transactional semantics only among transactions. Strong isolation guarantees transactional semantics between transactions and non-transactional code, in addition to transactions. Our algorithm assumes strong isolation.

- **Nested versus flattened transactions**: A nested transaction is one whose execution is properly contained in the dynamic extent of another transaction. The inner transaction sees modifications to the program state by the outer transaction. If the transactions are flattened, aborting the inner one causes the outer one to abort as well, but committing the inner one has no effect until the outer transaction is committed. Flattened transactions are easier to implement. The proposed algorithm doesn't make use of nested transactions.

- **Exceptions**: An exception thrown from within a transaction that leaves the scope of the transaction can either terminate or abort the transaction. The transaction can either be re-executed or terminated on an exception. The proposed algorithm doesn't require support for exception handling, since we don't expect language-level exceptions to be thrown from within transactions.

- **Transaction granularity**: This is the unit of storage over which a TM system detects conflicts. Most STM systems extend an object-based language and implement object granularity, i.e., conflicts are detected over different fields of an object. HTM systems generally detect conflicts at cache line/block or word granularity.

- **Direct and Deferred update**: TM implementations that use the "direct update" policy directly modify the shared object and rely on concurrency control to allow only one thread to update the object. They need some kind of rollback mechanism in case of a transaction failure. Deferred-update implementations update a private copy of the object and replace the actual object with the private copy at a later point.

- **Concurrency control**: Concurrency control is needed to deal with conflicts in a system where multiple concurrent transactions are permitted. Pessimistic concurrency control detects and resolves the conflict at the same time. Optimistic concurrency control permits the conflict detection and the resolution to happen at separate times as long as the resolution happens before the transaction is committed. This is also a way to guarantee forward progress, i.e., a transaction is guaranteed eventually to complete. Most TM systems are non-blocking in nature and offer one of the following guarantees:

    1. Wait-freedom: This is the strongest progress guarantees and ensures that all threads contending for a shared object will make forward progress in a finite number of their individual time steps.

2. Lock-freedom: This assures that at least one thread from the set of threads contending for a shared object will make forward progress in a finite number of time steps of any of the concurrent threads.
3. Obstruction-freedom: This is the weakest progress guarantee and only ensures that a thread will make forward progress in a finite number of its own time steps in the absence of contention over the shared object.

- **Early and late conflict detection**: A TM system can detect conflicts at three points:

  1. On transaction open, at the time of the first reference to an object.
  2. On validation, when a transaction examines the collection of objects it previously read or updated.
  3. On commit, when the transaction tries to commit.

- **Explicit vs Implicit HTM** The first step for an HTM is to identify transactional memory accesses. This is done via extensions to the instruction set. Based on these extensions, HTMs may broadly be classified into two categories: explicitly transactional and implicitly transactional.

  Explicitly transactional HTMs provide software with new memory instructions to indicate which memory accesses should be made transactionally (e.g., load_transactional and store_transactional), and they may also provide instructions that start and commit transactions (e.g., begin_transaction and end_transaction). Other memory locations accessed within the transaction through ordinary memory instructions do not participate in any transactional memory protocol. The examples of explicitly transactional memory are Herlihy and Moss HTM [38], Oklahoma Update [61] and the Advanced Synchronization Facility [6]. The advantages of an explicit HTM system are:

  – Explicitly identifying transactional accesses provides programmers with increased flexibility and may aid in reducing the size of a transaction's read-set and write-set.
  – Explicit interfaces also naturally allow transactional and non-transactional accesses to intermix.

  The primary drawback of an explicit HTM system is that using an explicit interface can require multiple versions of libraries; one for use inside transactions and one for use outside. Consequently, explicit interfaces can limit the reuse of legacy software libraries inside HW transactions. Therefore, such HTMs are more suitable for constructing lock-free data structures where the transactions are fairly small and limited in scope, and software reuse is not a priority.

Implicitly transactional HTMs on the other hand only require software to specify the boundaries of a transaction, typically demarcated by instructions like begin_transaction and end_transaction. They do not require software to identify each individual transactional memory access; all memory accesses within the boundaries are transactional. Examples of implicit HTM systems include the Azul HTM[62] and the newly announced Intel HTM [3]. The main advantages of an implicit HTM system is that implicit identification of transactional locations restricts instruction changes to the boundaries of a transaction. Therefore, these HTM systems can enable reuse of existing software libraries, and they typically do not require multiple code versions to be compiled. Most modern HTM proposals follow an implicitly transactional model.

Transactional memory implementations come in software-only flavors [55], hardware-only flavors [25, 3, 62] as well as hybrid implementations [22]. We confine our discussion to hardware transactional memory implementations, since we will be using one in the proposed Collie collector algorithm, and also since support for these is becoming commonplace [25, 3, 62].

1.3.2 Hardware Transactional Memory: Implementation Notes

HTMs must identify memory locations for transactional accesses, manage the read-sets and write-sets of the transactions, detect and resolve data conflicts, manage architectural register state, and commit or abort transactions. We now discuss some of the implementation techniques that HTM systems use to achieve these goals.

Tracking the Read/Write Set

An HTM must track a transaction's read-set and must buffer the transaction's write-set. Most HTM proposals are based on cache extensions that add an additional bit, the read bit, to each line in the data cache. A transactional read operation sets the bit, identifying the cache line as being part of the transaction's read-set. This is a fairly straightforward extension to a conventional data-cache design. Such designs also provide a way to clear all the read bits in the data cache instantaneously. Hardware caches enable HTMs to achieve low-overhead read-set tracking; however, they also constrain the granularity of conflict detection to that of a cache line. Additional access bits may be added to reduce the granularity for tracking. Write-set tracking is implemented in one of the following manners:

- Cache-based tracking: This extension requires the addition of a speculatively written state to be associated with the addresses written to. Since data caches are on the access path for processors, the latest transactional updates are automatically forwarded to subsequent read operations within the transaction; no special searching is required to locate the latest update. However, if the data cache is used to track the write-set, then care is required to ensure that an only copy of a line is not lost, e.g., in systems with write-back caches, where processors directly write to a location in the cache without updating main memory. A requirement is that the now-dirty cache line must eventually be copied into main memory before it is overwritten by a transaction's tentative updates. If we did not do this, we would lose the only copy of the cache line following an abort.

- Store-buffer based tracking: To avoid any modifications to the data cache, some HTM proposals add dedicated buffers to track read-sets and to buffer write-sets (Oklahoma Update [61]).

- Hybrid models are also possible; e.g., the Intel HTM [3] proposal and Rock HTM [25] both use the data cache to track the read-set while using the store buffer to track the write-set.

Zilles and Rajwar [69] studied the effectiveness of a data cache for tracking read-sets and write-sets for an HTM design with a 32KB 4-way set-associative data cache with 64-byte cache lines and found that such an HTM could frequently support transactions up to 30,000 instructions that operate on hundreds of cache lines; that was fairly large compared to common perceptions. However, in this kind of system, a transaction must be aborted whenever an overflow occurs. Such overflows occur due to set-associativity conflicts. Overflows can further be mitigated using a victim cache. A single victim cache increases the data cache utilization for transactions by 16% (increasing utilization to 42%).

Detecting Data Conflicts Using the MOESI Cache Coherence Protocol

Hardware cache coherence provides mechanisms to locate all cached copies of a memory location and to keep all such cached copies up-to-date. While numerous cache coherence protocols exist, most are either snoop-based or directory-based.

A write operation to a cache must keep all cached copies up to date. This usually involves invalidating stale (out-of-date) copies or (less commonly) updating those copies with the newly written value. If two processors simultaneously issue a write to the same location with different values, cache-coherence protocols ensure that all processors observe the two writes in the same order, with the same value persisting in all copies. In invalidation-based cache coherence protocols, the cache with the dirty copy of a line (i.e., the cache that has modified it with respect to memory) is responsible for servicing read requests from other processors which require access to the line.

We now discuss how the MOESI cache coherence protocol is overloaded to detect data conflicts. A similar approach can be used with other coherence protocols such as MSI. The

MOESI classification is based on the various states a cache supports, and the terms arise from the names of the cache states.

1. Invalid state (I): Address is not present in the cache.

2. Shared state (S):

   - Address is present in the data cache and it may also be present in other data caches, hence the shared nature of the state.
   - Because of the shared nature, if a processor needs to update the location, it must request write permissions from other data caches.
   - A processor cannot update an S state cache line by itself.

3. Exclusive state (E):

   - Address is exclusively present in the data cache and is not present in any other data cache.
   - A processor can typically update the E state cache line directly without asking for permission.
   - When it does so, the processor changes the state of the cache line to the modified state (M).
   - The E and S states share one common feature: in both cases, the cache line is clean, and the latest copy of the cache line also resides elsewhere, either in memory or another cache.

4. Modified state (M):

   - This is the latest copy of the cache line and is only present in this cache.
   - This data cache is responsible for servicing a subsequent request to this line.

5. Owned state (O): In the *O* state, the line is dirty but is also possibly shared with other caches. Since the line is dirty, this data cache owns the line and is responsible for providing this data when other processors request it. However, since it is shared, the processor cannot write the line without requesting write permission from other caches.

HTMs can utilize these existing MOESI protocols to detect conflicts: *any write request from another processor to a line in any state other than the I state that has been read transactionally is a data conflict: another processor is attempting to write to a line that has been read in a transaction. Similarly, any read request from another processor to a transactionally-written line in the M state is also a data conflict. In both these cases, the underlying cache coherence protocol automatically provides mechanisms to identify data conflicts.*

Resolving Data Conflicts

Once a data conflict has been detected, it must be resolved. Nearly all conventional HTM proposals perform eager conflict detection, aborting the transaction on the processor that receives the conflicting request, and transferring control to software following such an abort. This approach is simplest from the point of view of a hardware implementation. A software handler then executes and may decide to re-execute the transaction or to perform some kind of contention management. Notification of the causes of conflicts can be a valuable tool in debugging performance problems in software using TM: most HTM proposals provide a new architecture register for this purpose.

Managing Architectural Register State

In addition to memory state, software also has associated register state that is architecturally visible. Collectively, these architectural registers and memory combine to form a processor's precise state. When a transaction aborts, the register state therefore must also be restored to a known precise state, typically corresponding to just before the start of the transaction. Some HTM systems rely on software to perform such a recovery while other HTM systems build on existing hardware mechanisms to recover architectural register state. One straightforward approach for hardware register recovery entails creating a shadow copy of the architectural registers at the start of a transaction. This can be maintained in an architectural register file (ARF). This operation can often be performed in a single cycle.

We have discussed the design and implementation choices in building an HTM system. We now look at a sample HTM implementation, referred to as Speculative Lock Elision(SLE), that is similar to both the Azul HTM [62] and the newly introduced Intel™ Transactional Synchronization Interface(TSX) [3].

HTM Sample Implementation: Speculative Lock Elision

Speculative lock elision is a flavor of HTM that uses the data cache to track transactional read-set by extending each cache line with an access bit. The SLE implementation uses the store buffer to hold updates performed transactionally. Aborts require the store buffer's tentative updates to be discarded. Commits require making the store buffer's updates visible instantaneously. The hardware issues exclusive ownership requests for the addresses in the store buffer but without making the tentative updates visible. SLE also sets the access bit along with the M state when a tentative store accesses the data cache. This enables conflict checks against stores. SLE cannot initiate the commit sequence until all such requests have been granted. Once all required coherence permissions are obtained, the tentative updates must be made visible atomically. SLE stalls incoming snoop requests from other threads while it is draining tentative updates from its store buffer into the data cache. This ensures all updates are made visible instantaneously. This is guaranteed to complete in bounded time and without global coordination.

The key observation introduced by SLE was that a processor does not actually have to acquire a lock, but it need only to monitor that it remains available. The processor executes the critical section optimistically as if it is lock-free: two threads contending for the same lock could execute and commit in parallel without communication, so long as their executions were data-independent. It takes advantage of a key property of a typical lock variable: a lock-release's write undoes the changes of the lock-acquire's write. By eliding both of these operations, but monitoring the lock variable for data conflicts, the lock remains free and critical sections can execute concurrently.

1.4 Garbage Collection Performance

As stated earlier, a garbage collector generally goes through two phases: a *mark-phase* and a *recycle-phase*. The efficiency of the GC algorithm in recycling memory, i.e., "how fast it can identify and recycle garbage", directly effects the maximum application allocation rates it can support. An efficient, well designed stop-the-world collector can result in short application pauses. An efficient, concurrent collector can result in needing less amount of memory head-room to support a high allocation rate.

In order to increase GC efficiency, i.e., throughput, GC algorithms attempt to exploit the parallelism available in hardware by employing multiple task-threads to execute both the GC phases. Compaction follows the mark-phase and involves walking through liveness bitmaps to identify live objects and copying them to contiguous locations. Task-threads are assigned disjoint sections of the liveness bitmap heap to be compacted and there is very little synchronization needed between the parallel compacting task-threads. For this reason, GC algorithms that carry out compaction in parallel using task-threads scale very well as more task-threads are added to execute this phase.

Parallel marking algorithms, however, face a unique set of challenges. Marking involves two main aspects: a depth-first or a breadth-first walk through the heap graph and setting a liveness bit for each object encountered during the heap walk. A heap-walk is essentially a "pointer-chasing" algorithm and is not very amenable to prefetching techniques. In addition, task-threads used for marking have to walk the heap-graph in parallel and set the liveness bits in the marking bitmap using atomic primitives such as the compare-and-swap(CAS). Atomicity is needed while updating the marking-bitmap, since multiple objects that are marked live by different task threads could map to same word in the marking-bitmap. Parallel marking algorithms also need to implement load-balancing among the task-threads in order to ensure that all task-threads are kept busy and participate in the marking-phase. Load-balancing techniques add their own set of synchronization needs and these costs can often outweigh the benefit of load-balancing.

Concurrent marking algorithms, which mark the heap with concurrently executing application threads, have to deal with a changing heap graph as they continue to mark the heap graph. As noted earlier in section 1.2, they have to deal with changes to the heap graph after they have marked through it. They need to synchronize with application threads for this purpose and potentially re-mark sections of the heap.

We discuss the issues in implementing a scalable, concurrent and parallel marking algorithm; we identify the bottlenecks and propose a new non-blocking work-sharing algorithm between parallel marking task-threads in chapter 4.

1.5 Thesis Organization

This dissertation is divided into two distinct parts. The first part of the dissertation focusses on concurrent compaction techniques that provide low-cost and wait-free barrier synchronization mechanisms resulting in consistent application response times and higher throughputs. Chapter 2 of this dissertation introduces a new concurrent, compaction algorithm, the Collie collector, that provides the first of its kind, constant-cost, wait-free [39] read-barrier. The Collie collector makes use of the transactional hardware on the Azul platform [62]. Chapter 3 attempts to tease out a desirable quality of the Collie collector, i.e., the cheap, wait-free, read-barrier and outlines an implementation that doesn't require the use of transactional memory hardware. This algorithm is thus amenable to implementations on platforms that don't have a hardware transactional memory implementation available.

The second part of this dissertation focusses on improving the parallel marking performance of a concurrent collector. We identify the bottleneck in scaling parallel marking and propose a lock-free, non-blocking work-sharing algorithm for scalable parallel marking. Chapter 4 discusses the implementation and the performance of this algorithm within the state of the art, concurrent and parallel, C4 collector that runs on a Linux based X86 machine. Finally, chapter 5 reflects on the proposed algorithms and discusses the future direction of research.

Chapter 2

The Collie : A Wait-Free Compacting Collector

2.1 Introduction

Managed runtime environments, such as Java, that run the bulk of enterprise workloads, have been grappling with an abundance of cheap compute and memory capacity for sometime now. The objective has been to scale up in terms of memory and compute usage while maintaining response-time guarantees that are often in the 10s of milliseconds. There are some fundamental fault lines in the design of managed runtime environments that make it a challenge to achieve this dual goal. One such limitation is the usage of locks for synchronization in application and library code. The inability of current garbage-collection algorithms to scale to larger memory footprints without affecting application response/throughput is the second limitation, and the focus of this dissertation.

Traditional stop-the-world garbage-collection algorithms, where the application threads are stopped while memory is being reclaimed, have hit a wall as application heap sizes approach the 10s and 100s of gigabytes. This has resulted in the design and implementation of several concurrent garbage collection algorithms [21, 63, 44, 53, 10], all of which strive to recycle memory without stopping the application.

Controlling fragmentation is an essential part of any garbage-collection algorithm and is a requirement for real-time garbage collectors, since they need to be both time and space bound. Most real-time collectors [10] achieve this by either compacting the heap or by maintaining free-lists. Compacting collectors move live objects to contiguous memory locations and update the pointers to the obejcts' old locations to make them point to their new locations.

Real time garbage collectors need to be both time and space bound. Most real time collectors [10] achieve this by either compacting the heap or by maintaining free-lists. Controlling fragmentation is also an essential part of non real time garbage collection algorithms. Compacting collector move live objects to contiguous memory locations and updates the pointers to their old locations to make them point to their new locations.

Compacting the heap in a concurrent manner is a significant challenge because of the synchronization issues between the application threads and the compacting garbage collector. Most of these concurrent compacting algorithms [21, 44] suffer from a high rate of barrier execution, i.e., trap storms [63], expensive barrier code [21, 40] and a variable and sometimes unbounded amount of time [53] spent in the barrier code. These have an adverse effect on application throughput and cause unexpected jitters in application response times that can violate service-level agreements(SLAs). As noted in several papers [47, 63], application trends continue to push the previously assumed envelopes of accepted response times and heap usage. It is not uncommon for mainstream enterprise applications to expect consistent response times in the milliseconds, at the same time challenging notions of a small young generation that can be compacted within a stop-the-world pause. In the not very distant future, incremental approaches [10] and algorithms that are prone to trap storms, etc., will not be able to keep pace with application requirements. Clearly, achieving high throughput and consistent response times while concurrently compacting the heap continues to be the holy grail of garbage collection.

Our goal is to take a fresh look at the issues with concurrent compaction while keeping sight of the objectives listed above. In this chapter, we describe the Collie collector, a new approach to concurrent compaction. We have implemented it on the Azul JVM that runs on the custom system designed by Azul; previous collectors such as Pauseless Collector [21, 63] have been shipping on the same mature platform for several years now. The algorithm makes use of certain hardware features unique to the Azul platform such as a read-barrier instruction and the only shipping implementation of hardware transactional memory.

In the past couple of decades, transactional memory [38] has emerged as one of the alternatives to locks. Transactional memory strives to provide similar multi-address atomicity semantics as locks, while avoiding some of the pitfalls associated with locks such as priority inversion and convoying. Transactional-memory implementations span the spectrum from pure hardware implementations [62] and pure software implementations [55] to everything in between [22]. Chip manufacturers such as AMD have come up with proposals for constrained multi-address atomicity [6]. Most recently, Intel has announced support for an Azul style implicit hardware transactional memory(HTM) [3] for their next-generation Haswell processors. The semantics and the software API of Intel's HTM are identical to Azul's HTM implementation. IBM's Cell processor architecture also supports a variant of an implicit HTM. Thus, multi-address concurrency primitives providing speculative atomicity, i.e., hardware transactional memory, have truly arrived in modern architectures. The use of transactional memory in our algorithm allows it to be implemented using features such as Intel's TSX interface [3] or AMD's proposed "advanced synchronization facility" feature [6]. The contributions of this chapter are:

- We introduce the concept of an "object transplantation," the state associated with it, and a set of constraints based on the *linearizability* [39] framework that are necessary for correct concurrent transplantation.
- We introduce the first completely barrier-free compacting algorithm.
- We introduce the first wait-free [37], constant-time read barrier that is self-healing [21] and avoids the constant cost of the Brooks' [19] indirection barrier.
- We introduce the first concurrent compacting collector that makes use of hardware transactional memory support.
- We present an evaluation of our implementation on the Azul Vega platform, leveraging multi-address atomicity, a hardware transactional-memory feature of the Azul custom Vega chip. We compare Collie with with the Pauseless Collector [21] running on the same platform.

We set the framework necessary to discuss our algorithm, with a discussion on the issues involved with concurrent relocation in sections 2.2 through 2.4. Prior to discussing the proposed algorithm, we describe the platform features that are used by the algorithm in section 2.5. We follow that with a description of the algorithm in section 2.6, the implementation in section 2.8 and evaluation in section 2.10.

2.2 A General Look at Concurrent Compaction

Compaction of objects involves copying the reachable objects from a from-space into mostly contiguous memory locations in a to-space, replacing all pointers to from-space locations with corresponding pointers to to-space locations and reclaiming from-space memory for re-use. While the "from" and the "to" spaces could technically overlap in some collectors (such as the case in in-place compactors), this chapter is mostly concerned with compactors that use non-overlapping from-space and to-space address ranges.

Stop-the-world compaction is fairly straightforward, with the full set of compaction operations appearing to occur atomically from the mutator's perspective. However, concurrent compacting collectors[1] must deal with concurrent mutator activity. Concurrent compaction inevitably involves periods during object relocation, where multiple copies of the same object may exist at different addresses [63]. In this case, the collector must make sure that all threads see the same consistent state of the object, as specified by the memory model. Concurrent compacting collectors impose a set of constraints on the mutators to achieve the required consistency and usually enforce these constraints using different barrier mechanisms. There are two main options for maintaining consistency of object state during concurrent compaction. The first, which this chapter is mostly focused on, is to ensure that mutators can only observe one version of the object at any given time. For example, some collectors [13, 44, 21, 63] enforce a *to-space invariant*, where the mutator can only observe the to-space version of the object. The second option is to allow multiple copies to be visible to the mutators, with the collector guaranteeing that all reads from and writes to the contents of the logical object (which may interact with different copies) remain consistent as specified by the memory model[2]. For example, concurrent copying collectors such as the Sapphire collector [40] take this approach.

The *to-space invariant* is often enforced using a read barrier. When the barrier encounters a from-space reference, it will look up the corresponding to-space location and use it instead, copying the object to its to-space location if necessary. This ensures that all concurrent accesses to the object are performed to the to-space version of the object's contents. In some variations, the first thread to access a reference to a from-space object claims the rights to copy the object's contents to the to-space, while other concurrent accessors of the object wait on the copy to finish, in order to obtain the object's to-space pointer. In wait-free [37] copying variations, concurrent accessors may race and concurrently attempt to copy the same object's contents without waiting, with a "winning" copy being decided via an atomic operation. Collectors that use the Brooks style indirection barrier [19, 47] always traverse a forwarding pointer, which always points to the current (to-space) version of the object. Such indirection barriers are generally coupled with a write barrier to avoid propagation of from-space pointers. In contrast to indirection barriers, direct-access barriers enforce the *to-space invariant* on loaded reference values, e.g. Baker style [13] and LVB-style [63, 21] barriers. Varying implementations of direct-access barrier implementation exist including, hardware read-barrier instructions [21], in-lined code that checks for invariants [63] and virtual-memory protection techniques [21, 44].

Having garnered an intuitive feel for the requirements of compaction in general and for the consistency requirements specific to correct concurrent compaction, we now attempt to crystallize these into a set of constraints. We use concepts from *linearizability* [39] to highlight the requirements for correct concurrent relocation of an object. Within the linearizability framework, each operation appears to "take effect" instantaneously and the order of non-concurrent operations is preserved. Objects should go from a well-defined and consistent pre-operation

[1]This includes mostly concurrent compacting collectors such as the Baker collector [13] as well as the fully concurrent compacting collectors such as the C4 collector [63].

[2]According to the Java language spec [40], it is not necessary for the writes to different versions of the objects to be visible in the same order except in the case of *volatile* fields. However, ordering of reads and writes to non-volatile fields must still follow the memory model's ordering rules with regards to ordering operations such as crossing synchronization events and volatile access [41]

state to a well-defined and consistent post-operation state. Intermediate states are not defined and should not be visible. The linearizability framework maintains a history of operations on the object under consideration and defines a *legal sequential history*. A *sequential specification* for an object is a prefix-closed set of sequential histories for that object. A sequential history is *legal* if each object sub-history belongs to the *legal sequential specification* for that object. A *linearizable* object is one whose concurrent histories are linearizable with respect to the *legal sequential specification*. The *legal sequential specification* is obviously different for each concurrent data structure.

2.2.1 Referrer Sets and Object Transplantation

We introduce the notion of "object transplantation": from the point of view of an individual object, *any* relocating collector effectively "transplants" the object from one location to another. The transplantation of a single object includes the consistent moving of the object's contents to its to-space location (often referred to as relocation), as well as the replacement of all references to the object's from-space location with correct to-space references (often referred to as fixup). Transplantation of a given object is "complete" when both the contents move and all required reference replacement are complete.

Next, we define two terms for an individual object:

1. Referrer set: identifies the precise set of all object references (in the heap or in thread stacks/registers) that point to the object's location.

2. Conservative Referrer Set: identifies a set of object references (in the heap or in thread stacks/registers) that includes all references in the object's actual referrer set, but may also include other references that do not point to the object.

All relocating collectors transplant objects. Since each object transplantation must reliably correct all references in the object's referrer set before it is complete, the typical relocating collector achieves this reliability by using an extremely conservative global referrer set: a full from-roots traversal of all references in the object space being compacted, usually performed once in order to assure the completion of all object transplantations (e.g. copying collectors [40], or relocate + remap collectors [44, 63]). As a result such collectors only ensure that each object's transplantation is complete at the completion of a full from-roots traversal.

2.2.2 Constraints for General Concurrent Transplantation

It is useful to consider the key requirements that must be met for correct concurrent transplantation. We start with a general set of constraints that all concurrently transplanting collectors must adhere to. Section 2.3 will add further constraints to these, needed to support individual object transplantation capabilities. We define the notion of an "object's transplantation state" as:

1. The contents of the object itself.
2. The object's referrer set.

On a timeline, an object's transplantation state encompasses its state from the time it is marked for transplantation until transplantation is complete. In linearizability framework terms, a legal sequential specification for general concurrent transplantation is:

1. Until transplantation is complete, all observable copies of the object contents should remain consistent as specified by the platform's memory model.
2. Once transplantation is complete, no references pointing to the object's from-space location can exist in the object's referrer set.

A sequence of operations that violates this set of rules would result in an incorrect concurrent transplantation algorithm. We believe that all known relocating collectors (concurrent or not) adhere to these two rules. Baker [13] style algorithms, for example, meet all the linearizability requirements by assuring that threads always obtain the to-space pointer to the object, and then copying the object to the to-space if necessary, as object transplantation is assured to be complete at the end of the copy phase. Since copying is done by the first thread to claim rights to the object while other threads that try to access the object concurrently wait for the copy to complete, the object's observable contents remain consistent. The complete traversal of the reachable references during the copy phase ensures that no object transplantation set will include any from-space references. Another example is the Sapphire collector [40], which allows mutators to observe from-space references, and propagates writes to both versions of the object to make sure that they meet the consistency requirement. It will ensure that no references pointing to from-space locations exist at the end of a compaction phase, thereby meeting the second requirement.

2.3 Individual Transplantation

We introduce the distinction between individually transplantable and non-individually transplantable objects. In contrast to existing relocating collectors, the collector algorithms we introduce in section 2.6 use individual-object conservative referrer sets, which are significantly less conservative than a complete from-roots traversal. Where such individual conservative referrer sets can be established and maintained, we use them to perform individually complete object transplantations, which can be identified as complete well before the end of a GC cycle. For the purposes of the new algorithms we describe in this chapter, we introduce two additional sets of progressively restrictive requirements:

Heap-stable referrer set limitations: Per-object referrer sets can be established through tracing, so long as they meet and maintain certain requirements. We define a set that includes all heap locations containing references to an individual object and one that remains stable after tracing is complete, as a *heap-stable* set. Such sets, together with the global set of references in all thread stacks and registers, form conservative heap-stable referrer sets. Objects with valid heap-stable referrer sets after the end of the tracing pass are ones which, in addition to general concurrent transplantation limitations (above), adhere to an additional limitation on the legal sequential specification:

3. Once tracing begins, no references to the object may be written to the heap.

For objects that adhere to these limitations, the heap-stable referrer set will remain correctly conservative until the object's transplantation is complete. Objects that fail to adhere to these limitations at any point, from the beginning of tracing to the successful completion of their transplantation, are "hard" to establish a stable per-object conservative referrer set for, and are deemed to be non-individually transplantable.

Stable referrer set limitations: Individual object transplantation can be safely performed on objects that have a stable referrer set for the duration of their transplantation. Objects with a *stable* referrer set are ones which, in addition to heap-stable referrer set limitations, adhere to the two limitations on the legal sequential specification listed below:

4. At the beginning of transplantation, no thread stack/register references exist in the object's referrer set.

5. Once transplantation of an object begins, no new references may be added to the object's transplantation state until the transplantation completes.

Objects that adhere to these limitations exhibit a stable referrer set and can be individually transplanted in a single, object-specific "transplantation transaction." Since these limitations assure that no mutator can observe or modify the object contents during the transplantation operation, the transplantation state will appear to transition atomically from a pre-transplantation to a post-transplantation state. Objects that fail to adhere to these limitations at any point, from the beginning of tracing to the successful completion of their transplantation, are deemed to be non-individually transplantable. We discuss this further in section 2.6

In order to examine the implication of the legal sequential specification for an individually transplantable object, it is useful to discuss possible sequences of operations to which these rules apply: the operations that affect an object's transplantation state, as well as the operations used to establish “heap-stable referrer sets’ (primarily a trace operation). The set of possible operations on are:

1. Trace() : GC thread begins a trace to establish conservative heap-stable referrer sets.
2. Copy(ObjRefFrom, ObjRefTo) : GC thread begins copying object contents from its from space location to its to-space location.
3. UpdateRefs(ObjectFrom, ObjectTo) : GC thread begins to update all references to the object's from-space location to point to the object's to-space location.
4. Transplant(ObjRef) : GC thread begins the transplantation of the object (which consists of Copy() and UpdateRefs() operations).
5. Read/Write(ObjRef) : Java thread begins to read/write contents of an object through a reference to the object.
6. ObjRef=ReadRef(Object) : Java thread begins to read a reference to the object from a heap location.
7. WriteRef(ObjRef) : Java thread begins to write a copy of a reference into a heap location.
8. OK(Operation) : Acknowledgement of successful completion of any of the above operations.

It is important to note that the ReadRef() and the WriteRef() operations both add to the transplantation state of the object, since they create additional references (in the stack or the heap) that point to the object.

We now consider a sample heap graph, and will follow with some examples of operation sequences that violate our legal sequential specification. In figure 2.1, object A is being transplanted to a new location A'. The graph on the left hand side illustrates the object graph prior to transplanting object A. Object B and C have references to object A, while object A has a reference to object D. The pre-transplantation state associated with object A is represented as: $\{A, B_A, C_A\}$. The letter ‘A’ represents the contents of the object A, while a letter with a sub-script indicates a reference to the object.

The post-transplantation state is illustrated on the right hand side of figure 2.1, The post-transplantation state is represented as: $\{A', B_{A'}, C_{A'}\}$, ’). The pre and the post-transplantation states are the only legally observable states according to the sequential specification and all other states are invalid. Note: In our representation, A’ indicates a consistent object contents, and inconsistent object contents should not be visible to the mutator threads. Examples of

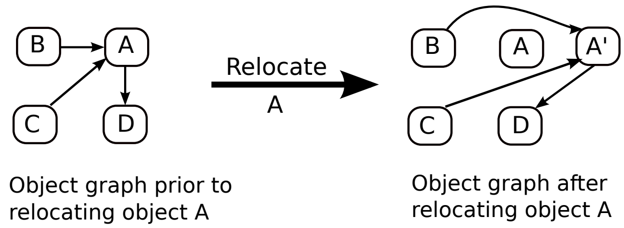

Figure 2.1: Object Graph Illustrating Object Relocation

sequences that violate the legal sequential specification for an individually transportable object: Consider a partial sequence of operations: *[...,Transplant(A), Copy(A, A'), stackRefA-1=ReadRef(A), OK(Copy(A, A')), OK(ReadRef(stackRefA-1)),...]* This sequence violates the legal sequential specification since ReadRef(A) creates an additional reference(stackRefA-1) to the object A after transplantation begins, thereby violating the stable referrer set limitations above, and rendering object A non-individually transplantable. Similarly the partial sequence listed below, where there is a write to the object being copied(via stackRefA-1) can result in inconsistent from and to copies of the object contents, thereby violating the first condition listed above: *[..., stackRefA-1=ReadRef(A), OK(ReadRef(stackRefA-1)), Transplant(A), Copy(A, A'), Write(stackRefA-1), OK(Copy(A, A')), OK(Write(A)), ...]*. Consider this sequence: *[..., stackRefA-1=ReadRef(A), OK(ReadRef(stackRefA-1)), Transplant(A), Copy(A, A'), OK(Copy(A, A')), UpdateRefs(A, A'), OK(Transplant(A)), Read(stack-RefA-1), OK(Read(stackRefA-1)), ...]*, this is an illegal sequence because the lifetime of stack-reference, stackRefA-1, spans the transplantation operation, and a reference to the object's from-space location exists after transplantation. On the other hand, the following sequence involving reads and writes of both object contents and references meets all the limitations for object A to be individually transplantable: *[Trace(), OK(Trace()), ReadRef(stackRefA-1), OK(ReadRef(stackRefA-1)), Transplant(A), Copy(A, A'), OK(Copy(A, A')), UpdateRefs(A, A'), OK(Transplant(A)), stackRefA-1=ReadRef(A'), OK(ReadRef(stackRefA-1), Read(stackRefA-1), ...]*

2.4 Barrier-Free Compaction

We focus our discussion here primarily on concurrent compaction, since concurrent marking is widely discussed [24, 21, 63, 23] and we believe that a highly performant, wait-free [37], and precise wavefront concurrent marker is essentially a solved problem with the C4's single-pass marker [63]. We now propose a completely barrier-free algorithm to individually transplant a specific object using a transplantation transaction. This barrier-free algorithm is not meant to be practical in cost or implementation. Instead, we will build on it by adding appropriate barriers and other mechanisms to describe the full Collie collector algorithm in later sections. We build on the notion of the transplantation state of the object. For the sake of discussion, we assume that the current and precise transplantation state of the object is available to the barrier-free collector when it starts to relocate the object. This implies that at that point in time, the version of the object reflects the consistent application state of the object contents, as per the runtime's memory model, and that the referrer set has precise information about all references to the object. We also assume that the referrer set includes only references that reside in the heap and that no references to the object exist in any mutator thread stacks/registers. For now, we have:

- a current and consistent version of the object contents.
- a precise referrer set to the object, that includes addresses of all the heap pointers to the object.

The collector starts the transplantation transaction, copies the object contents to its to-space location and updates all references in the referrer set to point to the to-space location of the object. Committing the transaction at this point would publish the pointers to the to-space version of the object and transplantation would be complete. In order for this commit to be correct, the following must remain true:

- ReadRef() operations should not occur concurrently with a committed transplantation transaction.
- Write() operations should not occur concurrently with the transplantation transaction if the writes could result in an inconsistent copy being created by the transaction.

These transaction rules could obviously be enforced by a simple (but expensive) lock scheme, where the transplantation transaction, mutator ReadRef operations, and mutator Write operations all use a lock on the object's from-space copy to synchronize their operations, but such locking would constitute barriers on mutator ReadRef() and Write() operations. We base our algorithm on the use of an Azul style implicit transactional-memory implementation, where all memory accesses are implicitly considered transactional once the transaction is started with a transaction-start operation, and are committed atomically using a transaction-commit operation. The transaction fails on potentially inconsistent concurrent writes to the transaction data set and throws an exception which is handled by a user routine. Concurrent reads of a transaction's "read" data set do not terminate the transaction; however potentially inconsistent concurrent reads of memory locations that are written to inside the transaction will result in a transaction failure.

Using these implicit transactional-memory semantics, the collector starts a memory transaction, "protects" the referrer set by writing to each of its references within the transaction, copies the object to its new location and updates the elements in the referrer set to the new location of the object. Committing the transaction at this point will publish both the to-space object contents and the reference values that point to the to-space location of the object and transplantation would be complete.

However, there is a time window between the time the object is marked for transplantation and the point in the transaction where each member of the referrer set has been protected by being written to within the transaction. Within this time window, the referrer set is not protected and ReadRef() operations can expand the referrer set without failing the transaction. To avoid this, we start the transaction inside a global safepoint and establish protection of the referrer set before allowing threads to continue past the safepoint.

Once allowed to run past the safepoint, mutator ReadRef() operations that happen concurrently with the transaction will abort the transaction, since the read is of one of the elements of the referrer set which belongs to the transaction's already-established write set. Since the mutator cannot obtain a reference to the object before the transaction completes without failing the transaction, attempts to concurrently read or write-to the object will also implicitly fail the transaction.

In essence, this algorithm requires a stop-the-world phase that builds a precise referrer set for the object and protects it within the starting transaction. Once the referrer set is established and protected, the remaining parts of the transplantation transaction can be executed and committed concurrently, since the implicit transaction semantics protect the transaction data set, including the referrer set. In the barrier-free compactor, such a safepoint would be for every object we wish to relocate, making it quite impractical. Algorithm 1 outlines the pseudo-code for relocating a single object in the barrier-free algorithm.

Algorithm 1 Barrier-Free Compaction

```
Start Safepoint.
Build Precise Referrer Set For Object.
Start Transaction
Protect Referrer set by writing to it.
End Safepoint
Relocate Object: Copy and Update Referrer Set
End Transaction
```

There are some key takeaways form this version of the algorithm.

- Barriers are not necessary during concurrent transplantation as long as the update of the referrer set and the object copy of the object contents are performed within a transaction.
- ReadRef() operations can pollute the referrer set outside of the transaction and need to be tracked or prevented. This can be done using either safepoints or barrier mechanisms.

2.5 Platform Support

We now take a look at some of the platform features used by the proposed concurrent compaction algorithm. We first present an abstract discussion on the functionality of the read barrier supported by the Azul hardware platform. We then describe support for global synchronization mechanisms in the virtual machine used by the proposed algorithm.

2.5.1 The Loaded Value Barrier (LVB)

The Loaded Value Barrier(LVB) is a read barrier supported by the Azul Vega hardware. It imposes a set of invariants on every object reference value as it is loaded from memory and made visible to the mutator, regardless of use. The two invariants imposed on all loaded reference values are:

- All visible loaded reference values will be safely "marked through" by the collector, if they haven't been already.
- All visible loaded reference values point to the current location of the safely accessible contents of the target objects they refer to.

An LVB can obviously encounter loaded reference value that do not meet one of these invariants, or both. In all such cases, the LVB will "trigger" and execute collector code to immediately remedy the situation and correct the reference such that it meets the required invariants, before making it visible to subsequent program operations.

LVB differs from a Brooks-style [19] indirection barrier in that, like a Baker-style [13] read barrier, it imposes invariants on references as they are loaded, rather than applying them as they are used. By applying to all loaded references, LVB guarantees no uncorrected references can be propagated by the mutator, facilitating certain single-pass guarantees.

LVB further differs from both Baker-style and Brooks-style collectors in two key ways:

1. LVB simultaneously imposes invariants both on the reference's target address and on the reference's marking state, facilitating both concurrent relocation and precise wavefront tracing [65] using a single fused barrier.

2. LVB ensures (and requires) that any trigger of either (or both) of its two invariant tests can be immediately and independently repaired by the mutator using self-healing behavior (see below), leading to efficient and predictable fast-path test execution and facilitating concurrent marking, concurrent relocation, and concurrent object transplantation characteristics.

2.5.2 Self-Healing

LVB is a self-healing barrier. Since the LVB is always executed at reference load time, it has access not only to the reference value being verified, but to the memory address the value was loaded from as well. When an LVB triggers and takes corrective action, modifying a reference to meet the LVB invariants, it will also "heal" the source memory location that the reference was loaded from by (atomically) storing a copy of the reference back to the source location. This allows mutators to immediately self-heal the root cause of each LVB trigger as it occurs, avoiding repeated triggers on the same loaded reference, and dramatically reducing the dynamic occurrence of read-barrier triggers. Each reference memory storage location will trigger "at most once" (discounting minute levels of atomicity races in the healing process). Since the number of references in the heap is finite, single-pass marking and single-pass reference remapping are both guaranteed in a straightforward manner.

Self-healing is uniquely enabled by the LVB's semantic position in the code stream, immediately following the reference load operation, and preceding all uses or propagation of the loaded reference value. This semantic proximity to the reference load operation grants the LVB access to the loaded reference's source address, which is required in order to perform the healing actions. Through self-healing, LVB dramatically reduces the dynamic occurrence of read-barrier triggering, making LVB significantly more efficient and predictable than both Brooks style and Baker style barriers, as well as other read barriers that will continue to trigger in the hot code path during certain GC phases.

2.5.3 Reference Metadata

The Azul Vega hardware supports the notion of metadata in pointers. We steal one address bit from the 64-bit address space; the hardware ignores this bit during addressing. This bit is called the Not-Marked-Through(NMT) bit and is used during the concurrent marking phase. The hardware maintains a register for the desired value for this bit. Object references with an NMT state that does not match the currently expected NMT value will trigger the LVB.

If, during an active mark phase, the LVB encounters a loaded object reference value with an NMT state that does not match the current expected state, the LVB will correct the situation by changing the NMT state to the expected value, and logging the reference on the collector's work list to make sure that it is safely traversed by the collector. Through self-healing, the contents of the memory location that the reference value was loaded from will be corrected as well.

At the beginning of the mark-phase, all live references are known to have an NMT value that matches the previous cycle's expected NMT value. As the mark-phase commences, the collector flips the expected NMT value, instantly making all live references "not marked through," arming the LVB to support single-pass marking. The collector proceeds to prime its work lists with roots and continues to mark until the work lists are exhausted, at which point all reachable objects in the heap are known to have been marked live and all reachable reference NMT states are known to be marked through.

2.5.4 Checkpoints

A *checkpoint* is a global barrier mechanism between the collector and the mutator threads, where the collector cannot proceed until all mutator threads have performed some action. In a *checkpoint* each mutator reaches a GC safepoint, does a small amount of GC-related work and then carries on. Blocked threads are already at GC safepoints; the collector performs the action on their behalf. In a stop-the-world pause, all mutators must reach a GC safepoint before any of them can proceed; the pause time is governed by the slowest thread. In a *checkpoint*, running threads are never idled and the GC work is spread out in time.

2.6 The Collie Collector Algorithm

We introduce the Collie Collector: a wait-free [37] compacting collector that uses bounded-cost read and write barriers combined with transactional operation to achieve concurrent individual-object transplantation. The Collie collector builds on the barrier-free compacting collector described in the previous section. It uses a concurrent wait-free tracer (performed as part of a C4-style [63] concurrent mark phase) along with a transactional transplantation phase to establish conservative, stable referrer sets. The collector develops conservative referrer sets for the vast majority of objects in the heap in a single tracing pass, relegating other objects to be transplanted into a newly introduced non-compacted *mirrored-to-space* using *zero-copy transplantation*. Collie replaces the requirement for a global safepoint in the barrier-free compacting collector with a combination of an enhanced LVB-style read barrier [63], a write barrier, a checkpoint operation [21], a copy operation, and a transactional transplantation operation.

2.6.1 Mirrored-to-space and Zero-Copy Transplantation

Before we identify the various mechanisms listed above as a replacement for the global safepoint, we introduce the notion of the "mirrored-to-space." The mirrored-to-space is a virtual address space that is identical in size to the from-space. Each from-space page is mapped or aliased to the same physical address as a corresponding mirrored-to-space page. A simple form of a mirrored-to-space is one where a single high-order bit differs between addresses in each mirrored-to-space page and its corresponding from-space page and where mirrored-to-space pages differ in this bit setting from all non-mirrored parts of the to-space. This form makes it easy to identify mirrored-to-space pages, and allows for easy "flipping" of roles between from-space and mirrored-to-space at the end of each GC cycle. The mirrored-to-space is logically considered part of to-space for mutator invariant and barrier test purposes, but is easily distinguished from "regular" to-space by operations that need to make such distinctions.

The purpose of the mirrored-to-space is to facilitate the *zero-copy, non-compacting transplantation* of objects from from-space to to-space. Since the object contents is inherently consistent across from-space object locations and their corresponding mirrored-to-space locations, such transplantation only requires the correcting of references pointing to the from-space location to cause them to point to the mirrored-to-space. This can be done in a straightforward manner using wait-free, bounded, and relatively cheap read barrier operations that easily fold into the LVB-style read barrier used by the Collie collector.

The Collie will attempt to individually transplant the vast majority of objects in the heap, compacting pages from which all objects were individually transplanted. However, the Collie is an aborting relocating collector and will abort the individual transplantation of objects that fail to meet the legal sequential specification limitations for stable referrer sets. Leaving such aborted objects in the from-space will break the to-space invariant, unnecessarily complicate the self-healing LVB-style read barrier, and increase its cost both statically and dynamically. Instead, the Collie performs a global (as opposed to individual) transplantation of all objects that fail to be individually transplanted, to their corresponding mirrored-to-space virtual addresses. Unlike typical global transplantation, no object copying operations are required for mirrored-to-space transplantation, making the transplantation no more costly for the mutator than concurrent mark.

2.6.2 Mutator Protocol

The mutator protocol includes a write-barrier and a read-barrier, as well as the implicit ability of each thread to individually come to a safepoint where the stack and registers can be scanned.
Write-Barrier: The Collie write-barrier intercepts all stores of references to the heap, identifying any object to which such references point as "non-individually transplantable" by setting

a "not relocatable" bit associated with the object's location. *The Collie write barrier directly satisfies the limitation required by heap-stable referrer sets*, thereby allowing the collector to reliably identify and use heap-stable referrer sets in a wait-free manner, without requiring a global safepoint[3].

Read-Barrier: The Collie collector uses an LVB-style read barrier which intercepts all reference loads from the heap. The LVB-style barrier is used to support both the concurrent tracer (in a manner similar to the Pauseless collector [21] during the concurrent mark phase), as well as Collie's concurrent compaction. Once the transplantation phase starts, the Collie read barrier ensures that any load of a heap reference that observes a from-space reference will attempt to atomically replace the from-space reference value in the heap with a mirrored-to-space reference value and then reload the reference from the heap location. Figure 2 shows the pseudo-code for the read-barrier trap handler. If the barrier succeeds in replacing the reference value in the heap, it has effectively atomically identified the object as "non-individually transplantable."[4] *The Collie read barrier directly satisfies the second limitation required by stable referrer sets as per section 2.3.* Combined with the write barrier and the pre-compaction checkpoint this quality allows the Collie to reliably establish and use stable referrer sets in both a cheap and wait-free manner, without requiring a global safepoint.

Algorithm 2 Pseudo-code for read-barrier trap code

```
function READ_BARRIER_CODE(address, oldValue, mirror)
    AtomicCompareAndSwap(address, oldValue, mirror)
    return *address
end function
```

2.7 The Collector Protocol

The Collie collector operates in two phases: *the mark-record* phase and the *transplantation* phase. There is no separate "fixup" phase and there is also no need to store forwarding pointers. The referrer set is essentially a per-object remembered set and is constructed during the mark

[3]While the write barrier will identify the object that the reference being written is pointing to as non-individually transplantable, it does not do so for the object that the overwritten reference was pointing to. As a result, a reference write may shrink a precise referrer set without recording the shrinking, making the previously established members of the referrer set conservative (they may be pointing to other objects). The conservative set remains correct, as it is inclusive of the precise set.

[4]If the replacement failed and the value had changed from the originally observed from-space reference, the newly loaded reference value is guaranteed to be a valid to-space reference, installed by a racing mutator or by a collector transplantation.

phase which already involves walking through the entire heap graph.

2.7.1 Mark-Record Phase

The mark-Record phase traces through all live objects in the heap, and serves the dual purpose of identifying the live objects as well as constructing per-object *heap-stable referrer sets*. The marking part of the algorithm is similar to that of the C4 collector [63].

The mark routine is overloaded to build the per-object heap-stable referrer set; when the mark routine reaches a "live" object as part of the heap graph walk, it already has the address of the referrer's reference field pointing to the object. Referrer set sizes can be capped at a static size, with objects that have more references than can fit in a referrer set being deemed "non-individually transplantable." The first time a referrer to an object is found, a referrer set is allocated and associated with the object. Objects that have more references pointing to them than can fit in a size-capped referrer set, or fail to allocate a referrer set, are deemed as "popular" objects and are identified as non-individually transplantable by setting a not-relocatable state associated with the object's location. Similar state can also be stored in the object header, and can be separately tracked to affect per-page relocation decisions[5].

At the end of marking, the heap-stable referrer set for each object includes, in addition to the per-object referrer set, the set of all mutator stacks and registers. A stack-scanning operation would still be required in order to qualify a heap-stable referrer set as a stable referrer set that includes only heap references.

This straightforward tracking of referrer sets has obvious memory overheads. Since the referrer set is nothing more than a per-object remembered set one can think of various compact representations of the referrer set that would allow convenient access at transplantation time. From empirical studies we know that most objects have only one or two references pointing to them, and as a result, we expect size-capping to be very effective in containing memory overhead. Similarly to other aborting compacting collectors such as Staccato collector [47] and the Chicken collector [53], avoiding transplantation of popular objects could potentially prevent some significant portion of the heap from being compacted. Empirically, popular objects make up a very small percentage of the live set for most applications. Not relocating popular objects could have some benefits as well, as noted in [23]. We discuss a workaround for the degenerate cases where the popular objects form a significant percentage of the live set.

[5] Compaction is performed at page granularity. While we are able to individually transplant the rest of the objects from a heap page that has a single non-individually transplantable object, the from-space page's physical resources cannot be freed if there are any objects in its corresponding mirrored-to-space page. For this reason and as an optimization, tracing can mark pages with popular objects in them as not-relocatable so that no compaction will be attempted on them in the compaction phase. In some cases, it may be desirable to still allow other objects to be individually transplanted away from some popular pages in order to increase the likelihood of the page being fully compacted in the future

2.7.2 The Transplantation Phase

The concurrent transplantation phase copies live objects to the to-space and eliminates references to from-space, allowing the from-space to be reclaimed. The collector compacts at a page granularity and uses the quick release technique [63] that allows it to free the physical memory backing each from-space page before references to it are eliminated, assuming it was able to copy all the live objects from the source page to the target page.

At the start of the transplantation phase the collector knows of a set of pages that are currently relocatable as well as a set of the per-object currently heap-stable referrer sets. The transplantation phase will individually transplant objects for which it can successfully maintain stable referrer sets. Objects that fail to be individually transplanted will be transplanted to the mirrored-to-space by the end of the GC cycle.

Collie's flavor of LVB-style read barrier serves two simultaneous functions: it assures that completed transplantation transactions are done using qualified stable referrer sets and supports the concurrent relocation of non-individually-transplantable into mirrored-to-space.

Pre-Compaction Checkpoint

In order to perform individual object transplantation, the transplantation phase needs to establish and maintain a stable referrer set for each individually transplanted object. The already-established heap-stable referrer sets include all references in the thread stacks and registers. The pre-compaction checkpoint establishes these stable referrer sets by scanning the roots and designating any object that they directly refer to as non-individually transplantable. The scanning of individual thread stacks is done in a checkpoint [21] without requiring a global safepoint.The pre-compaction checkpoint also arms the LVB-style read barrier, so that it will trigger on any load of a reference to a page that was currently relocatable at the start of the checkpoint.

After the checkpoint is passed, all objects that remain individually transplantable now have currently stable referrer sets. Any attempt to add references to such an object's transplantation state will render it non-individually transplantable. Thus, the pre-compaction checkpoint, combined with the read barrier's behavior directly satisfy the first limitation of *stable referrer sets.*

Concurrent Compaction

Once the collector establishes stable referrer sets and a set of individually transplantable objects using the concurrent tracer and the pre-compaction checkpoint, the Collie collector uses transactional operations to complete individual object transplantations. Each individual object transplantation starts by copying the object's contents of an individually transplantable object to a to-space location. The copying of object contents does not need to occur within a transaction because no mutator access to individually transplantable object contents can occur after the pre-compaction checkpoint without rendering the object non-individually transplantable. The collector uses transactional memory to update the referrer set:

1. A memory transaction is started. All memory accesses from this point until the transaction completes are done atomically with respect to other threads in the system.

2. Each reference in the currently stable referrer set is checked to verify that the reference does not currently point to the mirror virtual address of the object being transplanted. If any of the references fail this test, the transaction is aborted.

3. Each reference in the currently stable referrer set which points to the object's from-space location is replaced with a reference to the object's to-space location.

4. The memory transaction is committed.

The object transplantation is successful and complete when transaction commit succeeds. If the transaction fails or aborts for any reason, the object is rendered non-individually transplantable

Upon a successful individual transplantation the collector updates the stable referrer sets of any individually transplantable objects pointed to from the newly relocated object. This maintain the integrity of referrer sets that include references from this object. These referrer-set updates only needs to be synchronized against collector activity, and not against the mutator.

Through a combination of constraints applied by the LVB-style read barrier, the pre-compaction checkpoint, and the transplantation transaction, successful transactional transplantation operations are known to have satisfied all required limitations of stable and heap-stable referrer sets. They therefore result in correct, safe, and wait-free individual object transplantation.

Individually transplanted objects do not require a fixup phase, as each successful transplantation transaction has, by definition, already fixed all references to these objects. However, in order to complete the zero-copying transplantation of non-individually transplantable objects (to the mirrored to-space), the collector still needs to perform a full from-roots fixup traversal. Read-barriers are self-healing, no copying operations are involved, and all read barriers encountered by the mutator on not-yet-corrected references are both bound and "cheap." There is therefore no hurry to perform this traversal. Much like C4 [63], Collie simply rolls this fixup traversal into the next mark phase.

The collector invariant at the end of attempting to relocate an object is: *"all referrers to that object switch to using either the mirrored-to-space reference to the object or the to-space reference to the object"*.

2.8 Discussion

The Collie collector is a full-heap collector based on the notion of an object's transplantation state. The transplantation state is a distributed state; pointers to an object can be anywhere in the heap. The collector aggregates this state and relies on atomically updating it to the object's new location. The availability of hardware transactional memory on the Azul chip allows this multi-address update to be truly atomic. This simplifies the read-barrier trap code to a single atomic CAS followed by a read. The algorithm doesn't need any stop-the-world pauses; the pre-compaction checkpoint ensures that all mutator threads see the right protections before the start of concurrent transplantation. The transactional memory use in the algorithm is also very constrained, since the referrer set is the only thing that is updated inside the transaction. In the common case, the referrer set has two elements in it. This results in a high percentage of successful transactions and also makes the algorithm amenable to other similar implementations of multi-address atomicity such as the upcoming Intel TSX feature [3]. The aborting read-barrier has a flip side in that it can cause GC-induced fragmentation. This happens when a read-barrier aborts the copy of an object in a page that has been partially compacted by the collector. The collector can't reclaim the original source page or the target page. This however tends to be a rare event since the working-set of an application is a small fraction of the live-set. Also, the working-set generally correlates fairly closely with the root-set, which the collector doesn't attempt to relocate. This is similar to the issue faced by the Staccato collector [47]. Their suggested solution of increasing page density of these pages will work for us as well.

2.8.1 Trap Storms

A mutator faces a trap-storm when it attempts to access GC-protected objects resulting in triggering back-to-back GC traps. Most GC algorithms start a phase by "protecting" the program heap that includes the working-set of the application. This results in trap-storms frequently occurring at GC phase boundaries. Trap storms can also happen around application-phase switches that result in significant changes to the working-set. Trap storms result in mutators spending a significant ratio of a time slice in executing GC trap code. This adversely effects the application response times and throughput in those time windows, causing inconsistent application behavior.

The factors that determine the affect of trap-storms on mutator performance are:

- Working set: Trap-storms are primarily an affect of GC protecting the working-set of an application. Almost all concurrent compacting collectors protect the entire application heap that includes the working-set of the application. References to objects in the application working-set are generally found in the mutator execution context, i.e., the registers and the stacks. The Collie ignores objects with references in the execution context, i.e., the root-objects. This allows the Collie collector to significantly reduce the application exposure to traps. Algorithms such as the Compressor [44], start off their relocation phase by protecting the root-set and forcing the application threads to take traps and do the resulting GC-related work.

- Duration for which traps are enabled: As noted earlier, object relocation involves copying an object to its new location as well replacing the references to its old location with that of its new location. An object is protected as long as references to its old location remain in the heap. For Baker style copying collectors [13] and other concurrent compacting collectors such as the Compressor [44], this usually means that object transplantation is not complete until the end of the compaction phase. Algorithms such the Pauseless collector [21] and the C4 collector [63] prolong the transplantation by rolling the reference remapping of a GC cycle in with the mark-phase of the subsequent GC cycle. This results in prolonged periods where application threads can potentially run into trap storms. The Collie collector carries out complete individual object transplantation, which includes fixing up the reference in the object's stable referrer set, which significantly reduces the amount of time a mutator is exposed to potential trap events.

- Cost of a single trap: Almost all concurrent copying/compacting algorithms [13, 21, 44, 63, 7] that employ read-barriers to synchronize between the mutator and the concurrent moving collector use a flavor of the read-barrier that copies protected but yet unmoved, from-space objects to the to-space on behalf of GC. Most of them don't cap the amount of copying work that read-barrier trap code performs. In some cases such as the Compressor [44], the read-barrier does the additional work of remapping references in objects being copied. Most of these flavors of read-barriers also tend not to be wait-free; for example, the read-barrier employed by the Pauseless collector will spin-wait if it triggers on an object currently being copied by the collector. These result in expensive and variable-cost read-barriers. The Collie addresses both of these issues by employing a constant-cost, cheap read-barrier with no conditional statements in its body.

The Collie collector's decision to ignore relocating the root-set, its ability to maintain individual object stable referrer sets and most importantly its constant-cost, and cheap read-barrier significantly reduce the application exposure to trap storms.

2.9 Implementation

The algorithm describes a non-generational collector. However, it can be easily extended to be generational. While generational collection is an almost absolute requirement for production-quality garbage collector on modern hardware, incorporating one into the collector scheme is orthogonal to our work.

The algorithm is implemented in the Azul JVM [21] which is a production-quality JVM. While, the algorithm is truly concurrent and doesn't require any global safepoints, our current implementation does rely on safepoints.

2.10 Evaluation

The Collie algorithm aims to be a high-throughput collector that maintains consistent response times. The wait-free, constant-time, aborting read-barrier is one of the key reasons the collector allows the application to maintain response times that are within the SLA requirements. The collector also avoids relocating the working-set of the application thereby reducing the application exposure to jitter-inducing traps. We intend to evaluate and confirm that the collector is able to provide high responsiveness while being able to sustain high throughputs.

The collector has some unique facets such as the per-object referrer set and its property of not relocating the root-set and the popular-set. It also shares the aborting nature of the read-barrier with other collectors such as Staccato [47] and the Chicken collector [53]. We evaluate the effect of these design choices to ensure that they don't adversely affect the collector's ability to perform its primary function, i.e., recycle memory. We use a set of standard benchmarks for our experiments. We use the SPECjvm98 [60] suite, the Dacapo [15] suite of benchmarks as representative of client programs and the SPECjbb2000 benchmark that is more representative of a large server benchmark. We run all our tests on the Azul platform which is a 16-chip (384-CPU) Azul appliance with 128G of physical memory. The heap sizes are limited in each case, we highlight the details in the section below. We compare all our results to a modified version of Pauseless collector [21], a full heap collector running in the single-threaded mode. This allows us to make an apples to apples comparison, since both collector are implemented inside the same JVM, are full heap collectors, use the same LVB style read barrier (although the trap code differs), have identical heuristics that determine when GC cycles are triggered and run on the same hardware. This gives us a unique opportunity to shed light on the performance differential resulting from features that are unique to the Collie collector as compared to the Pauseless Collector [21]. We divide our evaluation into two sub-sections, we first discuss the effect of the Collie collectors unique properties in sec 2.10.1 and follow that with an evaluation of the Collie collectors ability to sustain high throughputs while consistently maintaining high response times.

2.10.1 Evaluating the Collector Properties

The Collie collector builds a per-object referrer set which is then atomically updated during object transplantation. The size of the referrer set is statically determined;popular objects can't fit all their referrers into this set. Page with popular objects are deemed popular pages and are not compacted. Thus the size of the referrer set directly affects the percentage of objects transplanted by the Collie collector. It directly affects the percentage of popular objects, it has also an effect on the percentage of aborted transplantations. A larger per-object referrer set allows the collector to compact more of the heap, thereby exposing it to more aborts as well.

We vary this variable in our experiments and measure its effect on various collector properties. Figure 2.2 plots the amount of garbage reclaimed as a percentage of the garbage found by the collector. Increasing the size of the referrer set allows the collector to reclaim more garbage. The percentage of memory reclaimed seems to top off for a referrer set size of 4 words. It is also worth nothing that for most benchmarks the biggest spike in garbage collected happens when the referrer set size goes from 1 word to 2 words. A referrer set size of 2 words allows us to reclaim over 80% of the garbage found in most cases. It is also interesting to note that except for fop and the antlr benchmarks, a referrer set size of 1 word already allows us to reclaim close to 70% of the garbage.

Figure 2.3 plots the percentage of popular pages in the live set found by the collector. The size of the referrer set is inversely proportional to the percentage of the popular pages; thus this metric drops quite a bit as we go to a referrer set size of 2 words. It is interesting to correlate figures 2.2 and 2.3; for smaller referrer set sizes of 1 or 2 words, we seem to have a high percentage of popular pages but at the same time we are also able to reclaim a very high percentage of the garbage found. This implies that popular pages are quite dense and most of the live-set seems to reside in these pages. Thus the collector won't benefit much from relocating these pages. These findings are in line with those of the G1 collector [23].

The Collie collector uses the aborting read barrier which has the flip side of creating GC induced fragmentation, so it is important to measure the percentage of pages that have their compaction aborted. Figure 2.4 plots this metric. It indicates a steady creep in the percentage of aborted copies as the referrer size increases. This is expected, since the collector attempts more transplants with the increasing size of the referrer set. This metric still is contained to a small percentage allowing the collector to reclaim most of the garbage found. All runs were done with a 512MB heap and the heuristic was tuned to do periodic GC's to force compaction.

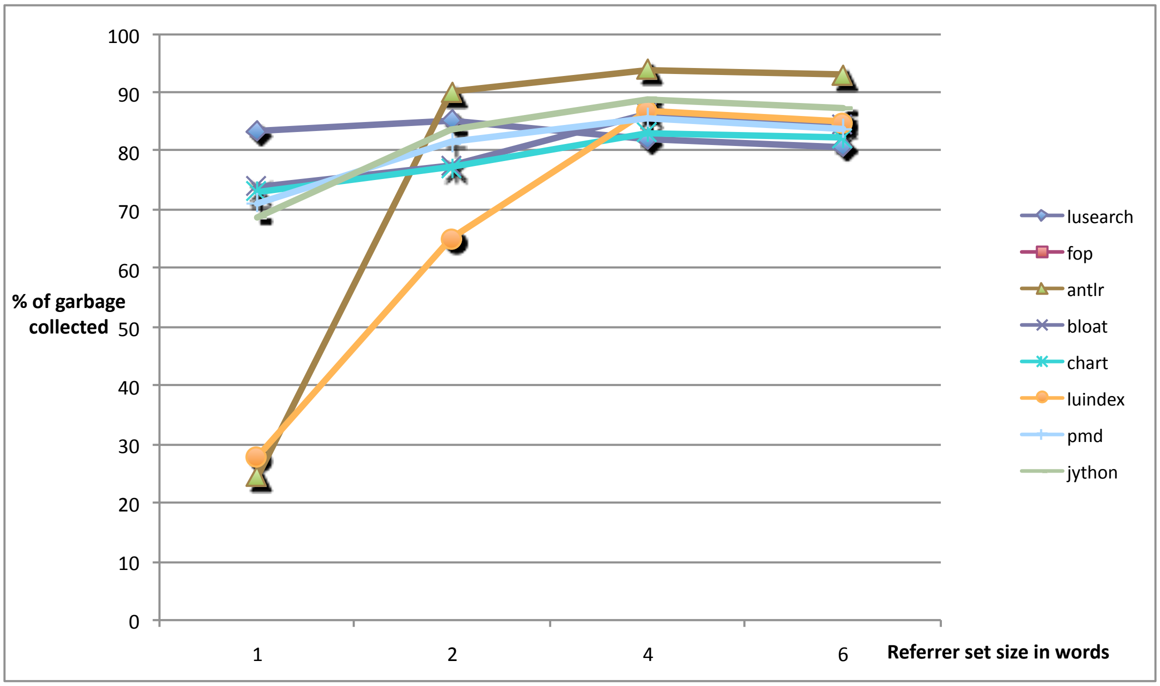

Figure 2.2: % of garbage collected increases as more pages are compacted with increasing referrer set size

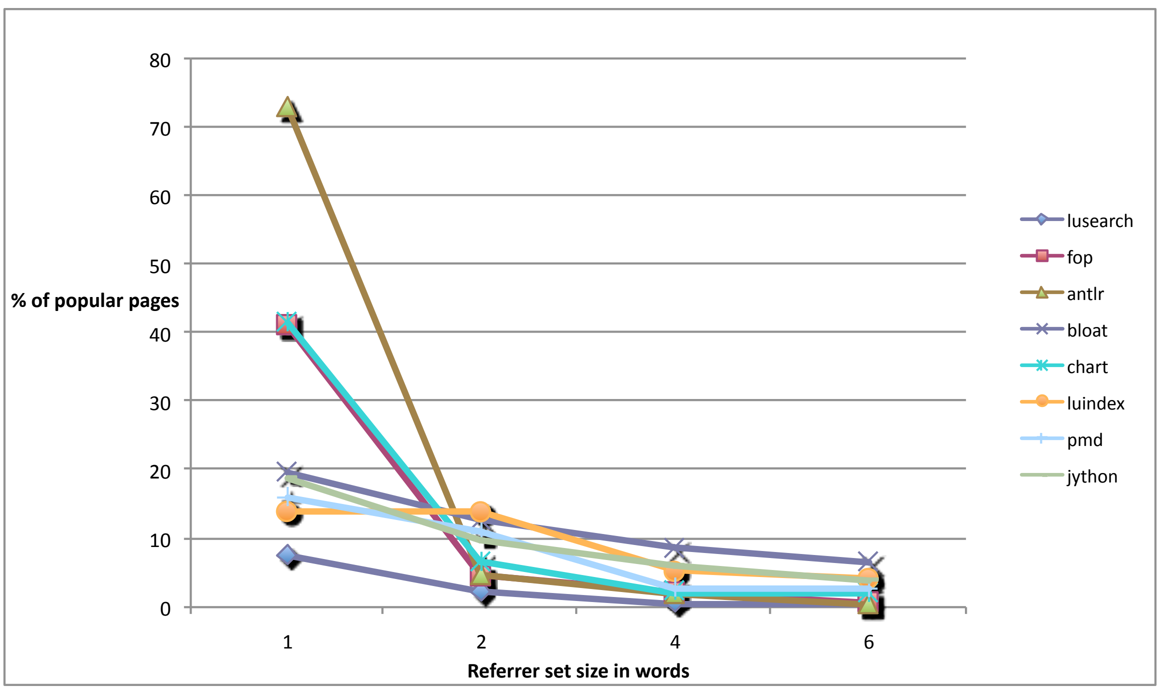

Figure 2.3: % of popular pages decreases with increasing referrer set size

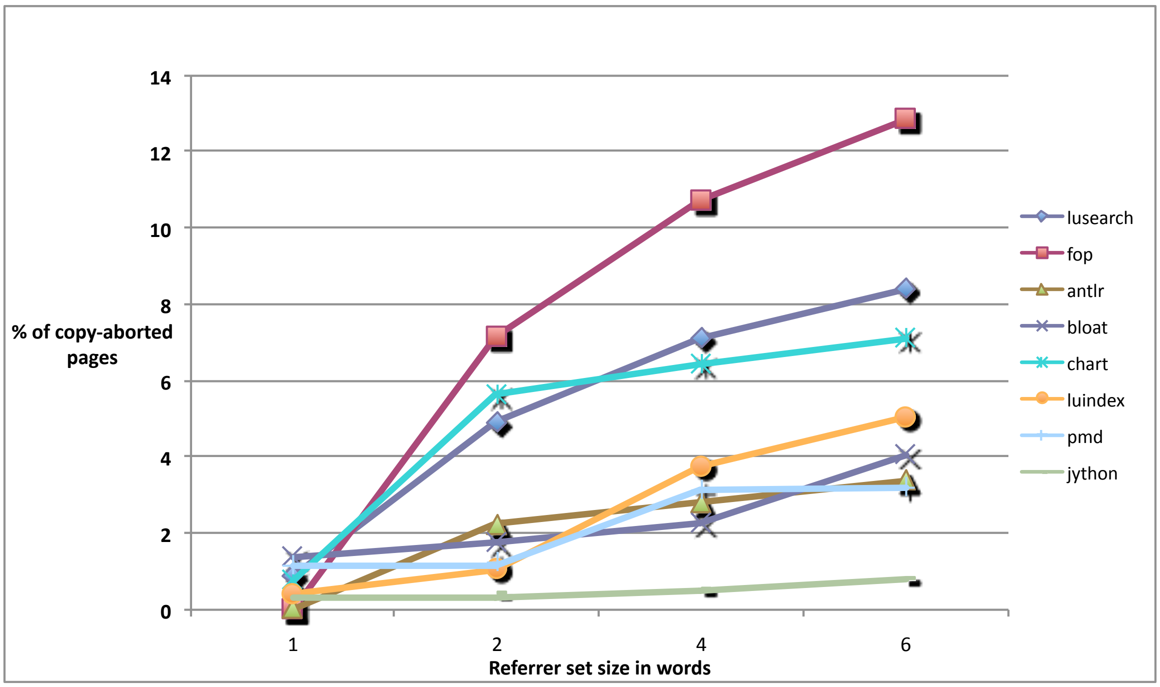

Figure 2.4: % of copy-aborted pages increases as an increase in referrer set size allows the collector to relocate more pages

2.10.2 Latency Evaluation

The Collie collector's main goal is to provide high responsiveness by sidestepping issues that affect other collector algorithms, such as trap storms and a variable amount of time spent in barrier code. We measure the minimum mutator utilization, (MMU) [11] metric to measure application responsiveness. We compare ourselves to the Pauseless Collector here. Since the collectors have a similar marking algorithm and identical phase switches, we focus on measuring the MMU during the concurrent compaction phase. The other collector-induced pauses are identical for the two collectors and have been studied in other places [21, 63].

Table 2.1 indicates the MMU percentages for the two collectors over various time windows ranging from 1ms to 200ms. Each column has two values in it, the first one reflects the MMU for the Collie collector for the given time window while the second number indicates the same metric for the Pauseless collector. The SPECjvm98 benchmarks were all run with 128M heap, while Dacapo benchmarks were run with a 512M heap size. As can be seen, the Collie collector does extremely well compared to the Pauseless Collector even in the 1ms time window. We believe that this is primarily attributable to the aborting read-barrier and the Collie collector's tendency not to transplant the working-set of the application. To reiterate: the MMU numbers for the Pauseless collector mentioned here are different from the previously reported MMU numbers, since we only measure the MMU during the concurrent relocation phase.

2.10.3 Throughput Evaluation

The Pauseless Collector [21] as well as the C4 collector [63] demonstrate very good throughput scalability as the heap sizes increase. The Collie collector which has a higher performant read barrier and is able to significantly reduce the application exposure to jitter inducing events such as trap storms, would be expected to do better. We use the SPECjbb2000 benchmark to measure application throughput differences between the two collectors. We ran these with 32 warehouses and a 30G heap. We were interested in teasing out the effect of the Collie collector's compaction algorithm on the application throughput. In order to do this we modified the heuristics for both the Collie collector and the Pauseless Collector so that we trigger back-to-back GC cycles. To further reduce the differences between the two collectors, the mark-phase was performed in a stop-the-world mode for both the collectors. Thus the entire measured throughput is attributable to the compaction phase of the collectors. Table 2.2 shows the results relative to the Pauseless collector, since absolute throughput numbers wouldn't have much meaning here. As we would expect, the Collie collector does improve over the Pauseless collector [21].

Table 2.3 shows the MMU for different time windows for the two collectors. The Collie collector again does significantly better than the pauseless collector providing consistently high MMU even in the sub-5ms time windows.

Table 2.1: MMU % at different time windows for the Collie/Pauseless collectors: Collie shows high MMU even at small time windows

Benchmark	at 1ms	at 5ms	at 20ms	at 50ms	at 100ms	at 200ms
compress	94/21	96/66	98/87	98/95	99/96	98/98
jess	85/35	95/56	98/68	99/77	99/85	99/92
raytrace	88/45	92/65	96/87	99/82	99/93	99/95
db	91/21	94/41	96/47	98/50	99/57	97/69
javac	87/24	93/56	98/71	99/79	99/86	99/88
mtrt	89/44	92/64	96/86	99/92	99/95	99/99
mpegaudio	87/45	90/66	92/87	99/92	99/95	99/97
check	91/41	93/64	94/85	97/93	99/95	98/97
batik	73/32	84/49	89/77	92/86	97/92	99/96
fop	78/21	85/31	88/62	92/79	96/82	99/84
luindex	84/39	92/80	96/94	98/97	99/98	99/99
sunflow	82/26	88/54	95/74	96/88	97/94	98/97
tomcat	80/0	86/29	93/73	98/83	92/92	99/95
lusearch	81/26	83/54	86/77	91/87	96/94	99/97

Table 2.2: SPECjbb2000 32 warehouse run, 30G heap: throughput comparison

Collection Algorithm	Throughput
Pauseless collector	1.0x
Collie collector	1.15x

Table 2.3: SPECjbb2000 32 warehouse run: MMU % at different time windows

Algorithm	at 1ms	at 5ms	at 20ms	at 50ms	at 100ms	at 200ms
Pauseless	22	45	64	80	81	83
Collie	81	86	89	91	95	96

2.11 Related Work

Cheng and Blelloch [12] designed and implemented a concurrent copying garbage collection with a bounded response time for a modern platform and also formalized the notion of minimum mutator utilization. The MMU metric tries to capture the amount of time spent by a mutator doing GC work. The Sapphire [40] collector is a concurrent copying collector that relies on a propagating write barrier to keep the two in-flight versions of an object in sync. The Chicken collector designed and implemented by Pizlo et al [53] is a variant of a concurrent copying collector where the write-barrier attempts to abort the copy instead of co-operating in copying the object. The Sapphire collector needs a blocking write-barrier to deal with writes to volatile fields, while the Chicken collector also needs a special-case barrier to deal with an application-level CAS. This reduces the blocking guarantees provided by the algorithm. The Staccato collector [47] is the closest to our work in that it uses an aborting read-barrier to obtain a wait-free barrier. However, their algorithm is tied to a Brooks style barrier, that is, an indirection barrier that suffers from the extra indirection and the extra word in the object header. More importantly, the Staccato algorithm cannot use any other flavor of the read-barrier since they rely on a single header word to store both the forwarding pointer and the relocation state of the object. The Collie algorithm is agnostic to the flavor of the read barrier. The literature records several other compacting collectors [44]. Compressor [44] is another one that uses page-protection techniques to achieve concurrent compaction but can suffer from trap storms. The Azul collectors, [21, 63] have been around for a while now and have focused on addressing the issue of scalable concurrent compaction while maintaining consistent response times. The use of transactional memory in GC algorithms is relatively new. [48] is a modified version of the Sapphire collector that relies on software transactional memory to make sure that concurrent updates to the object being relocated are not lost.

2.12 Conclusion & Future Work

The Collie collector is a concurrent compacting collector that uses a wait-free, copy-aborting read barrier. This work breaks new ground in several ways. We have introduced a first-of-a-kind treatment of concurrent compaction that defines the requirements for correct concurrent compaction. We introduce the notion of a per-object referrer set and are the first collector to build a concurrent compaction algorithm using hardware transactional memory. The aborting read-barrier that supports direct object references is a novel contribution as well. Each of these concepts can be applied individually to existing collectors for performance benefits.

We believe that we have only scratched the surface in terms of use of multi-address atomicity. Building the Collie algorithm using a pure software transactional memory implementation would be an interesting venture. Since heap graphs generally undergo minimal mutation between GC cycles, most referrer sets would remain fairly stable between GC cycles. This would allows us to experiment with techniques to incrementally maintain referrer sets in conjunction with write barriers. Per-object, referrer sets essentially store rich object graph information; referrer sets along with type information can also be used to augment leak-detection techniques.

Chapter 3

Wait-free Concurrent Compaction Using Single Word Atomicity

3.1 Introduction

The Collie collector discussed in chapter 2, supported a wait-free [37] read-barrier used as a synchronization mechanism between the mutator and the concurrent compacting collector. The constant-cost, wait-free semantics of the Collie read-barrier relied on the multi-address atomic update feature, i.e., transactional memory, available on the Azul Vega platform [62]. While transactional memory has seen a huge groundswell of interest in both the academic research community [38, 22] as well as in the industry [3, 6, 25], it hasn't seen widespread adoption in hardware platforms yet[1]. This chapter isolates the desirable quality of the Collie collector, i.e., its low cost, wait-free read-barrier and discusses a concurrent compacting algorithm that provides the same property while using a single-word atomic update primitive, widely available on most hardware platforms, viz., the *compare-and-swap*(CAS).

The Baker read-barrier [13], as well as those used by the Compressor [44] and the Pauseless collector [21], trigger on access to "protected" objects and copy the yet uncopied from-space object to the to-space. The read-barrier used by the Collie collector safely aborts copying an object that is marked for copying, but hasn't been copied yet. The copy-aborting nature of the read-barrier used by the Collie collector makes it cheap. In section 3.3, we discuss an aborting read-barrier flavor that is only slightly more expensive than the Collie collector, doesn't require transactional-memory support and still provides the desirable wait-free semantics of the Collie's read-barrier. We follow that with a discussion of different read-barriers and their performance characteristics in section 3.5. We end the chapter with a discussion on the performance charac-

[1] Intel has since announced support for an Azul-style implicit hardware transactional memory support in their next generation Haswell server processors [3]

teristics of the proposed algorithm. We first begin with a discussion on how a direct replacement of the Collie's use of transactional memory to update a object's referrer set, with a series of CAS instructions, violates the stable referrer-set requirements and use that as a platform to drive our new concurrent compaction algorithm.

3.2 Multi-word Object Transplantation State

Earlier in section 2.2.2, we defined the "object's transplantation state" as—

1. The contents of the object itself.
2. The object's referrer set.

On a timeline, an object's transplantation state encompasses its state from the time it is marked for transplantation until transplantation is complete. We also outlined a set of limitations based on the object's referrer set that need to be met in order to support successful individual object transplantation.

Stable referrer set limitations: Individual object transplantation can be safely performed on objects that have a stable referrer set for the duration of their transplantation. We reiterate the limitations on the legal sequential specification that objects must adhere to in order to maintain *stable* referrer sets:

1. Once tracing begins, no references to the object may be written to the heap.
2. At the beginning of transplantation, no thread stack/register references exist in the object's referrer set
3. Once transplantation of an object begins, no new references may be added to the object's transplantation state until the transplantation completes.

For the Collie, the first requirement is met by the write-barrier, while the second is met by the *pre-compaction checkpoint* described in section 2.7.2. The third requirement is met by the mutator's use of the aborting read-barrier and the Collie collector's use of hardware transactional memory to atomically update the members of the referrer set to the object's new location. Either the read-barrier succeeds in aborting the object transplantation or the collector's transaction succeeds in atomically updating all members of the referrer set. This provides correct transplantation semantics even when the referrer set has multiple elements in it; the multi-address atomicity provided by the transactional memory implementation ensures that there is no time window where a mutator can observe a pointer to the object's old location as well as one to its new location. The legal transactional requirements could potentially be

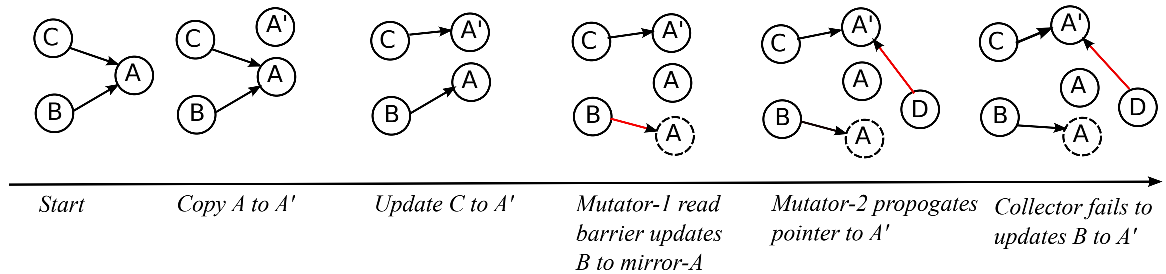

Figure 3.1: Stable referrer set requirements being violated when object transplantation is carried out using CAS

violated if the Collie collector were to use a series of CAS instructions to update the referrer set instead of transactional memory.

Figure 3.1 depicts the timeline of object A's heap-graph while it is transplanted using a series of CAS instructions and the manner in which it violates the stable set requirements. The state associated with object A prior to relocation is represented as: {A, B_A, C_A}. In this representation the letter "A" represents the contents of the object A, while a letter with a subscript indicates a pointer to the object. Object A has two elements in its referrer set: {B_A, C_A}.

Transplantation begins when the collector copies object A to its new location, A'. The collector then updates the pointer from object C to A' using a CAS instruction. At this point object B still points to the old location of object A. The mutator then accesses object A via the pointer in object B, triggering the read-barrier that updates the pointer in object B to point to the mirrored-to-space location of the object A. The mutator writes in the figure are indicated by red arrows and the mirrored-to-space version of object A is indicated by the dashed circle. At this point, the mutator believes that it has successfully aborted the transplantation of object A. However, a pointer to object A's to-space location, A', is visible to mutators and is propagated by another mutator in the consequent step. Object A's referrer set is now: {B_A, C_A, D_A}. This violates the third stable referrer set requirement listed above, since a new element has been added to the referrer set of object A while transplantation is still in progress. As discussed previously, this results in multiple copies of the same object in the heap, causing heap corruption.

The Collie's use of hardware transactional memory to update the object's referrer set ensures that mutators don't simultaneously see pointers to both the to-space and the from-space versions of the object being transplanted. The multi-address atomicity semantic of the transactional memory implementation allows it to detect a conflict when the read-barrier updates any element in its read/write set. However, the CAS is limited to detecting conflicts between updates to

<State>	<Forwarding pointer>	
45	41	0

Figure 3.2: 64-bit stateword used for maintaining per-object relocation state

a single address, which results in a time window where pointers to both the to-space and the from-space versions of the object are visible to the mutator threads.

In order to support a copy-aborting read barrier similar to that of the Collie collector using the CAS instruction, the *multi-word object transplantation state* needs to be distilled to a single-stateword. In the upcoming sections we describe an algorithm that achieves this.

3.3 Algorithm

The concurrent collection algorithm described in this section differs from the Collie collector only during the concurrent relocation phase. Similar to the Collie collector, it starts off with a mark-record phase, followed by a pre-compaction checkpoint. We will focus on the concurrent relocation phase of the algorithm and the corresponding mutator protocol.

As noted above, the collector maintains a single-stateword for each object and uses that to synchronize with the mutator read-barrier. The per-object stateword is stored along with the object's referrer set. The scheme for storing the referrer set is identical to that of the Collie collector, allowing easy access to the mutators. Figure 3.2 captures the format of stateword. The 64-bit stateword is split into two parts, the lower 41 bits[2] are used to hold the forwarding pointer of the object, i.e., its new to-space location, while the next 4 bits are used to hold the relocation state. There are three possible relocation states: *Allocated*, *Relocated* and *Copy-aborted*. We discuss the need for the forwarding pointer later in section 3.4. Figure 3.3 depicts the potential state transitions of an object being transplanted. We use this state diagram to outline the collector and mutator protocols in the following sections.

3.3.1 The Collector Protocol

Similar to the Collie collector, the single-stateword based collector establishes stable referrer sets and a set of individually transplantable objects using the concurrent tracer and the pre-compaction checkpoint. It then attempts to individually transplant all the objects that have been selected for relocation.

[2] This represents the virtual address range the JVM supports by default on the Azul Vega platform. This is not a limitation of the platform itself, that can support a full 64-bit virtual address range. This was done primarily for engineering reasons and can easily be extended.

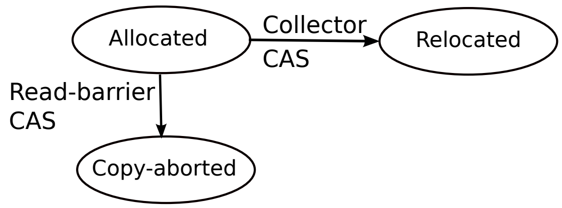

Figure 3.3: Object relocation state transitions

Each individual object transplantation starts by copying the object's contents of an individually transplantable object to a to-space location. Mutator accesses to individually transplantable object contents that occur after the pre-compaction checkpoint will render the object "non-individually transplantable". The collector follows the protocol outlined below to update the referrer set:

1. The collector first attempts to the update the object relocation state stored in the object's stateword to *Relocated*. As shown in figure 3.2, the new value of the stateword is a concatenation of the *relocated* state and the object's new location, i.e., the to-space pointer. If the CAS fails, the object transplantation is considered to be aborted and the collector moves on to the next transplantable object. On a successful CAS, the collector moves on to update the elements of the referrer set.

2. Each reference in the currently stable referrer set is now CAS-updated to the object's to-space location.

The object transplantation is successful and complete when all the elements of the referrer set are successfully updated to point to the to-space version of the object.

3.3.2 The Mutator Protocol: Read Barrier

Similar to the Collie collector, the single-stateword based collector, uses a LVB-style read barrier which intercepts all reference loads from the heap. Once the transplantation phase starts, the read barrier first attempts atomically to update the relocation stateword associated with the object to the *copy-aborted* state. If the read-barrier succeeds in transitioning the object relocation state to *copy-aborted* state, it attempts atomically to replace the from-space reference value in the heap with a mirrored-to-space reference value and then reloads the reference from the heap location. Figure 3 shows the pseudo-code for the read-barrier trap handler.

The mutator can fail to CAS update the stateword; this can happen if another mutator successfully aborts the object transplantation or if the collector succeeds in transplanting the

Algorithm 3 Pseudo-code for read-barrier trap code

```
function READ_BARRIER_CODE(stateWordAddress, objectAddress, oldValue, mirror)
    oldState ← <allocated><mirror>
    newState ← <copy_aborted><mirror>
    if oldState ≡ CompareAndSwap(stateWordAddress, oldState, newState) then
        CompareAndSwap(objectAddress, oldValue, mirror)
    end if
    forwardingPointer ← bottom_n_bits(*stateWordAddress)
    return forwardingPointer
end function
```

object. In either case, the "current" to-space pointer is left behind for the mutator in the bottom 41 bits of the stateword.

3.4 Discussion and Implementation

The single-stateword concurrent collection algorithm described in this chapter shares many of its features with the Collie collector. The implementation of the mark-record phase is identical between the two collectors. Both collectors also meet the same stable referrer set requirements outlined in chapter 2. The single-stateword collector doesn't make use of transactional memory, but rather distills the multi-word transplantation state to a per-object single-stateword. The mutator read-barrier code and the collector's transplantation code synchronize on this single-stateword using the CAS instruction. Object transplantation is aborted if the read-barrier succeeds in the CAS update of the stateword and vice-versa if the collector succeeds in updating the stateword.

The read-barrier code for the single-stateword based collector is slightly more expensive than the Collie's read-barrier, needing two CAS instructions in the case where the read-barrier successfully aborts the object transplantation. The read-barrier is wait-free [37] and its copy-aborting nature makes it much cheaper than the read-barrier used by the Pauseless collector as well as the Baker style read-barrier used in several other concurrent compacting collectors [44].

Forwarding pointer requirement:

Figure 3.4, shows the timeline for transplanting object A, which is currently being performed by the collector. Step-1 shows the starting heap-graph, with Object B and C both pointing to object A. Object A has two elements in its referrer set and the collector needs to update both these as part of the transplantation. We consider the case where the collector has already won the rights to transplant object A by successfully CAS-updating the relocation stateword of the object to the *Relocated* state. Step-2 illustrate that the collector has updated the object's

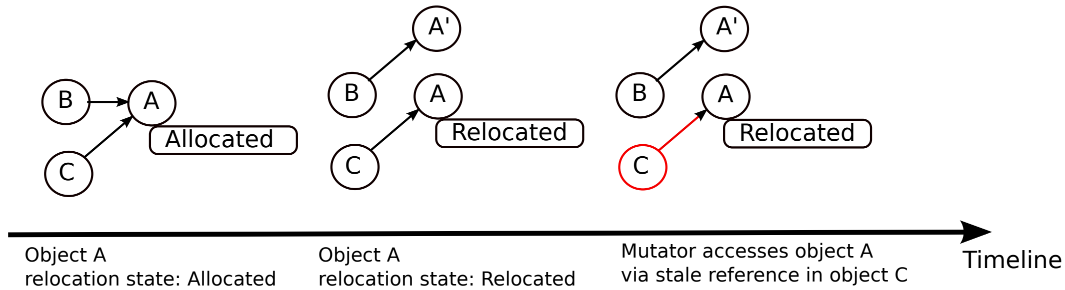

Figure 3.4: Stable referrer set requirements being violated when object transplantation is carried out using CAS

relocation stateword and has already updated the reference in object B to point to object A's new location. Object C still points to object A's old location. In Step-3, the mutator tries to access object A via the stale reference in object C that points to object A's "GC-protected" old location. The LVB read-barrier triggers, and finds the object's relocation state in the *relocated* state. At this point, the read-barrier trap code can't abort the object-copy since the collector already owns the right to transplant the object. The read-barrier trap code now knows that the object is in the midst of being transplanted, but doesn't know its new location. Since, the LVB semantics require that read-barrier trap-handling code always return a valid object pointer back to the mutator, the trap code will now have to wait till the collector gets around to updating the reference in object C to make it point to object A's new location, i.e.,A'. This breaks the wait-free [37] semantics that we want from the read-barrier trap code. To avoid this, we have installed a forwarding pointer in the object's relocation stateword. This allows the read-barrier to make progress by reading it from there instead of having to wait till the collector gets around to updating all the elements of the object's referrer set.

This is an issue for the single-stateword collector, because the mutator is allowed to see the "split" object transplantation state where some of the elements of the object's referrer set point to its to-space location while others still point to its old location. The Collie collector doesn't suffer from this since the multi-address atomicity provided by the hardware transactional memory implementation means that such a "split" object transplantation state is never visible to mutators.

Similar to the Collie collector, the aborted object-copy results in the "zero-copy" transplantation of the object to the mirrored to-space. Once an object is "transplanted," the forwarding pointer in the stateword points to the "current" location of the object, i.e., either to the to-space or the mirrored to-space. The forwarding pointer needs to be part of the stateword to avoid races between updating the object's relocation state and installing the forwarding pointer.

The use of the forwarding pointer obviates the need for maintaining and updating the referrer set. The single-stateword collector can take the approach used by other concurrent compacting collectors that use a forwarding pointer [21, 63, 44, 13, 53] by implementing a separate "fixup" phase where the collector does a heap walk to update the object pointers to their current locations. The "fixup" phase can be rolled into the mark phase of the upcoming GC cycle, an approach taken by the Pauseless collector [21, 63]. Our implementation of the single-stateword collector maintains referrer-sets in order to retain as many of the Collie collector's properties for an apples-to-apples comparison between the two algorithms.

3.5 Read Barrier Comparison

The design of the read barrier trap code used to support concurrent compaction is critical to the application performance. Both the Collie and the single-stateword collector described in this chapter use wait-free read-barriers for synchronization between the mutator and the concurrent compacting collector. The Collie additionally has a constant-cost read-barrier, the predictability of which makes it even more desirable in real-time situations [53]. We now discuss some of the read-barrier flavors that have been published in literature over the past decade, and compare their qualities with the qualities of the read-barrier used by the Collie and the single-stateword collector.

We first look at the read-barrier trap code used by the Compressor(2006) [44], a concurrent compacting algorithm. The compressor uses virtual-memory protections to protect objects that have not been relocated yet. The read-barrier triggers on access to one of these objects. The compressor's read-barrier trap code first attempts to obtain rights to "fixup" the virtual page it trapped on via an attempted CAS on a per-virtual page, status word. The read-barrier can lose this CAS to another mutator or the collector, in which case it has to "wait" till the other mutator or the collector finishes the "fixup" on the protected page. Thus the compressor's read-barrier is not "wait-free." In addition, the compressor's read-barrier has multiple conditional slow paths in its code. The read-barrier can be required to "fixup" an entire page of protected objects, which can potentially include copying these objects and un-protecting the protected virtual page via system-call. These make the slow path of the compressor's read-barrier very expensive. As expected, their application allocation rate, which directly correlates to the MMU, falls to 0 in the 0–10ms time window. Their MMU's continue to stay in the low 10's in the 10–200 ms time window for almost all benchmarks [44].

The Pauseless collector(2005) [21] and the C4 collector(2010) [63] use similar read-barrier mechanisms to support concurrent compaction. The Azul read barriers cooperate with the concurrent compacting collector by copying protected objects to their to-space locations if they haven't been relocated yet. A mutator thread that traps on an object that is currently being relocated, either by another mutator or the collector, has to "wait" till the relocation finishes.

Thus the Azul read-barriers are also not "wait-free." In addition, the read-barrier has variable cost, since it is required to copy objects that haven't been relocated yet. Depending on the size of the object, this could potentially cause a very long "pause" for the mutator. The Azul barriers are better than the one used by the Compressor [44], since they only have to deal with the object they trapped on and don't make any system calls. This reflects in slightly better MMUs compared to the Compressor [44]. However, the MMU for the Azul collectors still hovers in the low 30's for the 0–50 ms time windows [63, 21].

The Chicken collector(2008) [53] uses both a read and a write barrier during the collector's concurrent compaction phase. The read-barrier is a Brooks-style indirection barrier [19] that looks up a forwarding pointer in the object header. An object that has not been relocated by the collector has a forwarding pointer that points to itself. The forwarding pointer will point to the object's new location if it has been relocated by the collector. A Brooks-style barrier has a constant indirection cost, i.e., the read-barrier trap code has to pay the cost of the indirection lookup even when the object has not been relocated by the collector. The write-barrier used by the Chicken collector is a copy-aborting style barrier. It uses a protocol similar to that used by the single-stateword collector described in this chapter. However, since the copy-abort is done by the write-barrier instead of the read-barrier, the collector has to deal with application threads reading pointers to a relocated object's old location. This requires the collector to "handshake" with the application threads to fixup the from-space pointers in the stacks and registers of the application threads. This "handshake" is generally a stop-the-world handshake, resulting in zero MMU during the handshake itself.

In addition, application compare-and-swap operations are supported by a designated CAS barrier that extends the Chicken collector's standard write barrier. The collector also requires a special barrier to deal with pointer-equality tests that redefine $a == b$ to return true even when a is a from-space pointer and b is a to-space pointer to the same object (or vice-versa). CAS operations on objects must be redefined in a similar manner, resulting in a lock-free, rather than wait-free, CAS implementation [53]. The Chicken collector was implemented in the .NET framework. The authors and do not report MMU or application allocation rates. They report custom metrics on custom benchmarks, making it hard to compare their performance with the performance of either of the collectors discussed in this dissertation. However, since their algorithm is "lock-free" as opposed to being "wait-free", it is are prone to time windows where at least one of the application threads will have to "wait."

The collector algorithms proposed in this dissertation take a holistic approach to improving MMU. Both algorithms significantly reduce the cost of a single read-barrier trap. In addition, both algorithms avoid relocating the application working set, thereby limiting the number of read-barrier traps that are triggered. Finally, both algorithms use per-object referrer sets and update them to an object's new location, as part of the object's transplantation. This further reduces the time duration for which references to "protected" pages are kept around in the heap. These properties of both the concurrent, compacting algorithms discussed in this dissertation result in the high MMU numbers observed.

3.6 Performance

Our implementation of the single-stateword collector shares most of Collie's characteristics, i.e., the copy-aborting read-barrier, per-object referrer sets and "stationary" working sets. We focus primarily on the effect of the single-stateword collector's read-barrier. Similar to the Collie, the main goal of the single-stateword collector is to provide high responsiveness. We measure the minimum mutator utilization, i.e., MMU [11] metric to measure application responsiveness and compare the performance of the single-stateword collector to that of the Collie collector and the Pauseless Collector. Since all three collectors have similar marking algorithms and identical phase switches, we focus on measuring the MMU during the concurrent compaction phase.

Table 3.1 indicates the MMU percentages for all three collectors over various time windows ranging from 1ms to 200ms. Each column has three values in it, the first reflects the MMU for the single-stateword collector for the given time window while the second and the third numbers give the same metric for the Collie and the Pauseless collectors, in that order. The SPECjvm98 benchmarks were all run with 128M heap, while Dacapo benchmarks were run with a 512M heap size.

As expected, the single-stateword collector does better than the Pauseless Collector across all time windows. However, the Collie does better than both the collector's, since its read-barrier is the cheapest among the three. We measure the throughput of a 32 warehouse, SPECjbb2000 run using the same measurement methodology as outlined in section 2.10.3. Table 3.2 shows the relative throughput for the three collectors. As expected, the single-stateword collector does better than the Pauseless collector [21], while the Collie performs the best.

3.7 Conclusion & Future Work

The single-stateword collector described in this chapter attempts to bring the low-cost, wait-free, copy-aborting read-barrier to stock platforms that don't support hardware transactional memory. The multi-word object-transplantation state is distilled into a single per-object stateword. The per-object stateword is used for synchronizing between the concurrent compacting collector and the mutator, with both using a widely available atomic update primitive, viz., the CAS instruction, to update the stateword. As expected, the proposed collector does better than the Pauseless collector, while doing slightly worse compared to the Collie. However, unlike the

Table 3.1: MMU % at different time windows for the Single-stateword/Collie/Pauseless collectors

Benchmark	at 1ms	at 5ms	at 20ms	at 50ms	at 100ms	at 200ms
compress	80/94/21	83/96/66	89/98/87	96/98/95	98/99/96	98/98/98
jess	73/85/35	79/95/56	82/98/68	88/99/77	93/99/85	97/99/92
raytrace	69/88/45	74/92/65	89/96/87	91/99/82	93/99/93	96/99/95
db	53/91/21	59/94/41	73/96/47	79/98/50	83/99/57	89/97/69
javac	67/87/24	64/93/56	75/98/71	83/99/79	91/99/86	96/99/88
mtrt	52/89/44	61/92/64	83/96/86	94/99/92	96/99/95	98/99/99
mpegaudio	60/87/45	73/90/66	89/92/87	94/99/92	96/99/95	99/99/97
check	66/91/41	71/93/64	89/94/85	94/97/93	96/99/95	95/98/97
batik	53/73/32	59/84/49	79/89/77	88/92/86	94/97/92	98/99/96
fop	50/78/21	65/85/31	72/88/62	83/92/79	89/96/82	96/99/84
luindex	67/84/39	89/92/80	92/96/94	94/98/97	93/99/98	97/99/99
sunflow	62/82/26	67/88/54	77/95/74	91/96/88	93/97/94	95/98/97
tomcat	47/80/0	79/86/29	84/93/73	87/98/83	90/92/92	93/99/95
lusearch	59/81/26	63/83/54	81/86/77	83/91/87	91/96/94	95/99/97
SPECjbb2000	55/81/26	69/83/54	83/86/77	89/91/87	95/96/94	99/99/97

Table 3.2: SPECjbb2000 32 warehouse run with a 30G heap: throughput comparison

Collection Algorithm	Throughput
Pauseless collector	1.0x
Collie collector	1.15x
Single-stateword collector	1.08x

Collie collector, the single-stateword collector doesn't require a per-object referrer set, since it maintains a forwarding pointer.

The single-stateword collector can be built on current hardware while the Collie can be built on the next generation X86 hardware that will support a hardware transactional memory [3] implementation similar to that supported by the Azul Vega platform.

Chapter 4

Scalable Concurrent and Parallel Mark

4.1 Introduction

Parallel garbage-collection algorithms have been around for a while [35] and are best suited to take advantage of modern multi-core architectures. Parallel, stop-the-world collectors are effectively able to reduce GC-induced application pauses. Parallel, concurrent collectors support higher application allocation rates and more consistent application response times [63]. Parallel GC algorithms [31, 63] use several threads to execute both the "mark" and the "memory reclamation" phases[1]. In this chapter, we focus on concurrent, parallel marking algorithms.

Parallel marking algorithms use multiple threads to mark through the live-set in the program heap. Each thread starts with a stack primed with references to root objects. Each object pointer in the marking-stack is popped out, the object it points to is marked, and the references to its children are then pushed back onto the marking-stack. This is the typical tracing loop which proceeds until no new references to unmarked objects are found. In case of parallel marking schemes that use a mark-bitmap with a bit for every potential object address, the task-threads need to use atomic instructions, such as the compare-and-swap(CAS) instruction, to update the mark-bitmap. This is required in order to avoid losing simultaneous updates by different task-threads to the same word in the mark-bitmap by different task-threads. Byte-maps allow us to avoid the expense of atomic operations, with a byte in the marking byte-map for every potential object address, resulting in a 8-fold increase in memory usage. This is not very desirable, especially with the increasing heap sizes of current enterprise applications. Locating the mark bit in the object header allows us to avoid atomic updates in stop-the-world mark algorithms.

[1]Memory reclamation can be done via sweeping or by compaction. We only focus on GC algorithms that have a separate "mark" phase. We don't discuss mark-copy algorithms that tend to merge the two phases.

However, concurrent marking implementations are generally required to atomically update the mark bit in the object header, since the header word is used for other purposes such as storing the object hash-codes. Mark bitmaps also have several other advantages compared to locating the marking bit in the object header. Marking bitmaps store marks much more densely. The collector does fewer writes to the object in a design using a mark-bitmap, resulting in fewer dirty cache lines. Also, since objects live and die in clusters [43], mark-bits can be tested and cleared in groups. For these reasons, mark-bitmaps are prevalent in implementations of most commercial garbage-collection algorithms [24, 63, 23].

At the outset, the parallel mark phase generally tries to distribute the work evenly among the task-threads. Most parallel marking implementations also rely on load-balancing algorithms [31, 30, 29, 58, 57, 68], where the primary focus is to ensure that there is enough work for every processor involved in marking. The initial work assignment and the subsequent work-sharing algorithms tend to be oblivious to the memory location of the objects being processed by the respective task-threads and hence are classified as "processor-centric" algorithms [42]. The current work-sharing approaches achieve an improvement in marking throughput; however their scalability doesn't quite approach the ceilings of the available parallelism [57, 14].

A purely processor-centric work-sharing scheme in the mark phase, along with the clustered nature of the mark-bitmap, results in multiple task-threads updating a single word in the mark-bitmap. In this chapter, our premise is that contention caused by atomic updates to the mark-bitmap is one of the primary bottlenecks to the scalability of current work-sharing schemes in the mark phase. This results in increased cache traffic, and cache pollution that adversely effects the marking throughput. In case of concurrent mark algorithms this will also affect mutator performance. We discuss our evaluation methodology and use it to study the performance of our baseline algorithm in section 4.3

We propose, implement and analyze a new work-sharing scheme that reduces contention among the task-threads during updates to the mark-bitmap in section 4.4. Our scheme makes a cache line in the mark-bitmap into the basic unit of work sharing. This ensures that only a smaller, tunable number of task-threads update addresses in the mark-bitmap that map to the same cache line. Our algorithm is non-blocking and lock-free. It is important to note that our scheme is orthogonal to load-balancing schemes such as work-stealing [31, 30], which can be easily overlaid on top of our proposed schemes. Similarly, our proposed scheme is also amenable to prefetching algorithms. The proposed scheme is a first of its kind "hybrid" work-sharing algorithm that tries to reduce cache traffic while keeping all involved processors busy.

Parallel marking algorithms use marking stacks that are thread-local to the collector's task-threads to store the work that is generated during the traversal of the heap graph. The marking stack stores references to "gray" objects, i.e., objects that are potentially not "marked through" yet. Mutator threads are the other source of tracing work in concurrent [63, 21] as well as on-the-fly marking schemes [27, 9]. In these schemes, mutator threads are provided with thread-local marking stacks where they collect references to potentially “gray” [26] objects. The work accumulated in the mutator marking stacks is transferred to the collector’s task-threads using different schemes. Domani et al. [27] use double-ended queues for this purpose. Azatchi [9] uses soft handshakes to drain this work. The C4 collector [63] uses soft handshakes, referred to as checkpoints, to transfer this work to a global pool of marking stacks; the collector’s task-threads then acquire these stacks from the global pool in a thread-safe manner and mark through the references in these stacks.

We note that since this work is generated by mutators when they reference these objects, there is a high likelihood that these objects continue to be present in the processor/core cache. In section 4.6, we describe and evaluate a scheme to distribute the mutator-generated marking work among the collector’s task-threads, that is cognizant of the system topology [2].

Our proposed algorithms have been prototyped within the C4 collector [63], that runs on a Linux-based X86 machine. We would like to point out that the proposed algorithms are independent of the C4 collector algorithm and can be applied to other mark-sweep or mark-compact GC algorithms. The contributions of this chapter are:

- We describe a methodology to evaluate the scalability of a concurrent, parallel marking algorithm and highlight a bottleneck therein.
- We identify the two primary sources of work for task-threads during the concurrent mark phase, i.e., work generated by the task-threads themselves and the work generated by mutator threads, and propose separate algorithms to improve the parallel “marking throughput” in each case.
- We describe and evaluate a new non-blocking and lock-free work-sharing algorithm for concurrent, parallel marking
- Finally, we describe a “processor and cache topology” aware work-sharing scheme for mutator-generated tracing work.

4.2 Related Work

The *mark phase* consumes the bulk of the time in a typical GC cycle [18]. This is primarily attributable to the poor temporal and spatial locality properties of a pointer-chasing algorithm. There is a large body of work on improving the performance of the mark phase. There are two main approaches: techniques that focus on improving the single-threaded performance of a marker and those that aim to improve the overall “marking” throughput in a parallel environment with multiple marker threads.

Prefetching techniques [18, 20] focus on improving the single-threaded performance of a marker by trying to ensure that the object about to be marked through is found in the processor cache. Prefetching-on-gray [18] fetches the first cache line of an object as that object is grayed, i.e., added to the marking stack. This technique relies on the timing of the prefetch and is susceptible to a prefetch being ineffective because it is either too early or too late. Cher et al. [20] add a FIFO stack in front of the traditional marking stack, prefetching an object when a reference to the object is first added to the FIFO stack. The fixed-size FIFO allows the timing of the prefetches to be tightly controlled, thereby increasing the chances of an effective prefetch. Garner et al. [32] restructure the tracing loop by enqueuing "edges" instead of "nodes". All non-null references to child objects are pushed onto the marking stack and the "test-and-mark" operation is performed when these references are popped from the stack. This sequence results in the marker performing back-to-back operations on the same object, increasing the benefits of having the object in cache via prefetching.

Parallel marking algorithms reduce tracing time by having multiple threads walk through disjoint sections of the object graph. Siebert [58] studies the potential parallelism available to marker threads, and points out that most applications have rich enough data structures to support a high level of parallelism. Several algorithms have tried to exploit this, most of which have been classified as processor-centric algorithms [42]. These algorithms rely on "work-stealing" [57, 30, 31] based load-balancing techniques to ensure that all participating threads are kept busy.

Endo et al. [30] use a per-thread marking stack and a steal-queue to implement their load-balancing algorithm. Each thread periodically allows up to half of its work to be stolen, by transferring it from its private marking stack to the steal-queue. Once a thread is done draining the work in its marking stack, it attempts to obtain work from the steal queues of other task-threads. Steal-queues are protected by locks, with an optimization to avoid waiting on an already-held lock. Flood et al. [31] use a per-thread fixed-size deque, in conjunction with a lock-free work-stealing algorithm based on Arora et al. [8] to achieve load-balancing. Their approach has very little synchronization overhead. Once a task-thread is done with the work in its marking stack, it steals a single piece of work from the marking stack of another task-thread. Siebert [57] also use a work-stealing mechanism in their implementation of a parallel and concurrent real-time garbage collector. In this case, the granularity of of work stolen is much larger, with a thief thread attempting to steal an entire marking stack. Each algorithm trades off between the synchronization costs involved in work-stealing with the amount of work stolen. Wu et al. [68] propose a scheme where each task-thread uses a single-writer, single-reader channel to push work out to other task-threads. In a system with M task-threads, each task-thread has an array of $M-1$ circular buffers used for work-sharing. The single-writer, single-reader scheme allows them to avoid the expense of atomic operations during load balancing.

Parallel marking algorithms with fixed marking stacks have to deal with potential stack overflow. Flood et al. [31] implement a global per-class overflow set. The "overflow" objects of the class are maintained in a singly linked list chained through the "type" field in the class structure. Ossia et al. [52] use the notion of gray packets to share work among task-threads, where stacks full of references to "gray" objects are pushed onto a global linked list of stacks. They use this global pool, both for "work-sharing," as well as to deal with stack overflow. The C4 collector [63] uses a similar global pool of marking stacks to deal with stack overflow.

Parallel marking algorithms employing work-stealing have to implement a termination condition that ensures that all participating threads continue to stick around and assist their sibling threads even after they are done with their local work. Flood et al. [31] achieve this by having the task-threads spin on a non-zero status word that has a bit for each participating thread. Each thread sets its bit in the status word when it starts working. One can also use a counter-based termination condition, where task-threads spin on an atomically updated counter that indicates the number of active task-threads. Wu et al. [68] use a dedicated thread to detect termination.

Currently, there aren't any "memory-centric" [42] load-balancing algorithms used in parallel marking that we know of, although these are quite common in parallel copying algorithms. Our algorithm proposes a first-of-its-kind, hybrid work-sharing scheme that attempts to reduce the contention during atomic updates of the mark-bitmap, while keeping all participating threads busy.

4.3 Evaluating a Parallel Marking Algorithm

The primary goals of a parallel marking algorithm are:

1. To reduce the time it takes to mark the heap graph.

2. To make optimal use of the hardware resources assigned for marking. This generally implies the use of work-sharing schemes to ensure that all the involved processors/cores are ideally marking disjoint sections of the heap graph in parallel.

3. To keep the synchronization costs resulting from load balancing to a minimum.

We begin by describing our baseline concurrent, parallel marking algorithm. We then describe our measurement methodologies and highlight the bottlenecks in the algorithm.

4.3.1 Baseline Algorithm

The C4 collector is a concurrent, parallel, generational compacting collector with a tunable set of task-threads dedicated to collecting the garbage in each generation. The marking algorithm for both generations is identical. The C4 collector's use of the LVB-style read-barrier to support concurrent mark has been discussed in detail elsewhere [63, 21]. We won't focus on the read-barrier mechanism and the generational aspects of the collector from here on, since it is orthogonal to our current discussion.

Each task-thread has a LIFO style marking-stack used for the depth-first traversal of the heap graph. The mark phase starts with the marking stacks of the task-threads primed with references to root objects. Each task-thread first begins by draining its marking stack. For each reference in the stack, it enumerates the pointers to child objects, filters null references, marks the yet-unmarked child objects and pushes the references to these child objects back onto its marking stack. The marking stack can overflow during this process and when this happens, the marking stack is added to a global overflow pool of marking stacks. The global overflow pool is maintained as a linked list of marking stacks. This is similar to the gray-packet approach used by Ossia et al. [52]. The global overflow pool of marking stacks is a non-blocking data structure; stacks are added to and acquired from the global pool using atomic instructions such as the compare-and-swap instruction(CAS). Empty marking stacks are maintained in a separate pool; hence our algorithm doesn't suffer from the "ABA" [8] problem. Each task-thread also gets a FIFO queue to enqueue prefetch requests [20].

Once a task-thread has drained its local marking stack, it tries to acquire a "full" marking stack from the overflow pool. If the overflow pool is found to be empty, the task-thread attempts to steal a single reference from another task-thread's marking stack. The work-stealing algorithm is a "steal-one" algorithm, the details of which we don't discuss here in the interest of space and since it is orthogonal to this chapter's work. The overflow pool of marking stacks and the work-stealing algorithm are part of the processor-centric load-balancing algorithm used by the C4 collector. Since every GC algorithm uses a different processor-centric load-balancing scheme, our goal is to study the performance scalability of the common aspects of these GC algorithms. For this reason, we don't focus on the load-balancing aspects of the C4 collector in this chapter.

The task-threads maintain a shared counter of the number of active threads, which is a single word in memory updated using the CAS instruction. This counter is used to calculate the termination condition [31]. The pseudo-code for the baseline algorithm is outlined in fig 4.

4.3.2 Evaluation Methodology

A collector's task-thread performs a series of operations during the mark phase, such as atomically setting a bit in the mark-bitmap, pushing and popping the marking stack, and executing load-balancing operations that vary depending on the algorithm being used. Testing whether an object is already marked and attempting to atomically update a word in the mark-bitmap is common to most parallel marking algorithms. We describe a methodology for measuring the

Algorithm 4 Baseline Marking Algorithm

```
shared over_flow_pool
shared marking_stack[N]
shared task_thread_count = M                                        ▷ M task-threads
shared working_count
me = myThreadId
function ACQUIRE_WORK()
    if (!marking_stack[me].is_empty()) then
        return
    end if
    if acquire_from_overflow_pool(marking_stack[me]) then
        return
    end if
    i = 0
    while (i < task_thread_count) do
        if (ref = steal_one(marking_stack[i])) then
            return
        end if
        i=i+1
    end while
end function
function DRAIN_MARKING_STACK()
    while (!marking_stack[me].is_empty()) do
        ref = pop(marking_stack[me]);
        mark_through(ref);
    end while
end function
function MARK_THROUGH(ref)
    child_pointer_list = ref.child_pointer_list();
    while child_pointer_list.is_not_empty() do
        child = p.load()
        if (child! = NULL) then
            marking_stack[me].push(child)
        end if
    end while
end function
```

Algorithm 4 Baseline Marking Algorithm (continued)

```
function PUSH(ref)
    if (ref.test_and_mark()) then
        marking_stack[me].add(ref);
    end if
    if (marking_stack[me].is_full()) then
        add_to_overflow_pool(marking_stack[me])
    end if
end function
function MARK_TASK()
    while (true) do
        if (acquire_work()) then
            inc_working_count()
            drain_marking_stack()
            dec_working_count()
        end if
        if ((working_count == 0)) then
            break;
        end if
    end while
end function
```

performance of the test-and-mark operation in section 4.3.2. In section 4.3.2, we describe our approach for measuring the contention resulting from the collector's task-threads atomically updating the mark-bitmap. We first describe the benchmarks we use for our studies, the platform that we used for our runs and the heap-sizing policies in section 4.3.2.

Framework Details

We use the DaCapo (version 9.12-bach) [15] and the SPECJVM-98 [60] suite of benchmarks for our experiments. All experiments are done on a 2-socket Intel Westmere-X5680 based system, running the 2.6.34 Linux kernel with 98GB of memory. Each socket has 6 hyper-threaded cores, making it a 24 "core" machine.

The size and the shape of the program live-set has a direct bearing on the performance of a marking algorithm. In order to study the performance of a marking algorithm, it is important to ensure that GC cycles are triggered during all program phases. This in turn guarantees that the marker has to deal with different shapes and sizes of the live-set. The maximum heap allocated (-Xmx) to an application doesn't directly affect the time spent by the GC algorithm in marking. It can however, impact the GC heuristics that determine how often GC cycles are triggered. We avoid this by tweaking our GC heuristics to trigger periodic GC cycles. This ensures that the marker is forced to work during different program phases. The change in GC heuristic doesn't impact the size and the shape of the live-set kept live by "strong references". We use a time based promotion policy, hence triggering frequent new generation cycles doesn't change the live-set in the old-generation.

Periodic garbage-collection cycles can, however, end up frequently clearing soft/weak references that hold caches, which can result in changes to program behavior. In order to avoid this, we don't clear out soft/weak references and run with a generous enough heap size to allow the benchmarks to run to completion without running out of memory. Also, the C4 collector implementation differentiates between marking-through of strong references versus weak/soft references. We use this feature of the C4 collector to measure performance only during the marking of strong references. We note that strong references constitute the majority of the references in all the benchmarks; hence our measurement window covers bulk of the mark phase. However, we did run with different heap sizes and found no impact on the measurements, which is in line with the observation from Garner et al [32]. Similar to Garner et al [32], we run all our experiments with a generous heap size that allowed us to run to completion. We found that a heap size of 1GB was enough to run all the benchmarks. We also use their approach to condense the huge amount of data produced by running a total of 22 benchmarks (14 from DaCapo and 8 from SPECJVM-98) and report the geometric mean for each metric for the full set of benchmarks across both collector generations.

The C4 collector is a concurrent generational collector and has a unique inter-generational concurrency property [63], i.e., both the new and the old generations can be collected concurrently. The C4 collector assigns a tunable set of task-threads for each generation. Since our experiments are done on a 24-core box, we don't allow the total number of task-threads across both generations to exceed that number. Also, for measuring the performance of our baseline algorithm and that of the proposed work-sharing algorithm during the mark phase, we have the C4 collector's marker operate in a stop-the-world mode. This ensures that all the processor cores in the system are available for the marker's task-threads and our numbers are not polluted by scheduler events.

Measuring the Performance of "Test-and-Mark"

We chose the *average number of processor cycles* as the metric to measure the performance of test-and-mark for an object. On the X86 platform, we can obtain this by reading the time-stamp counter [1] prior to and after a test-and-mark operation for an object and recording the delta between the time-stamps. Other hardware platforms offer similar "tick" counters as well. While this approach gives us a high-resolution measure of marking an object, it is also an expensive operation. The "RDTSCP" [1] instruction, used for reading the time-stamp counter on the X86 platform, is a serializing operation and waits for prior instructions to finish executing before reading the time-stamp counter. We would like to amortize this cost by measuring the test-and-mark cost in batches. However, in the traditional tracing loop outlined in Fig 4 there are several other operations between any two "mark" attempts, such as the "push", "pop", "null-check" and "enumerate pointer fields" operations. We tweak the tracing loop in order to tease out the test-and-mark operations and execute a batch of only these operations.

We place a fixed-size FIFO stack in front of the marking stack to hold references to unmarked objects. Non-null object references are first pushed onto this FIFO stack and a batch of test-and-mark operations is started when this buffer is full. As part of the batch operation, the task-thread executes a test-and-mark operation on each reference in the FIFO stack. The time-stamp counter is read prior to starting the batch operation and at the end of the batch operation. The delta in the time-stamp counter values and the number of references processed are accumulated in thread-local counters.

An object that was marked live by the task-thread as part of the batch needs to be marked through, and hence a reference to that object needs to be pushed onto the main marking stack. The cost of this push should not be included in the cost of the batched test-and-mark operations. In order to exclude it, we use the highest bit in each stack slot of the FIFO buffer to indicate whether the particular object was marked as part of the batch operation. Since each stack slot is a single 64-bit memory word and the virtual address space is restricted to 48 bits on the X86 platform, we can safely use the highest bit of each slot for this purpose. At the end of the batch operation, we push the references that are in the slots with highest bit set, onto the main marking stack. This filtering out of references that were not marked live by the task-thread under consideration, reflects the original semantics of marking and doesn't pollute our measurements.

This approach is similar to that used by Cher et al. [20] for implementing the FIFO-based prefetching algorithm. The pseudo-code is outlined in Fig 5. Using the time-stamp counter for measurements is susceptible to scheduling events that occur in the measurement window. This can result in large, undesirable spikes in the measurement. On a multi-core machine, this is exacerbated since the time-stamp counters are processor-local and not necessarily in sync. For our measurements, we use a FIFO stack with 10 entries. Using a small fixed-size FIFO allows us to tightly control the measurement window and avoid scheduling events in the middle of our measurement window for the most part. This approach filters out most of the scheduling events within the measurement window, but not all. We discard measured values that exceed a static threshold that matches the scheduler time quantum on our platform (10ms). Such values occur extremely rarely. This gives us a pragmatic approach to obtaining high-resolution measurements.

Fig 4.1 plots the processor-cycles to mark a reference against the number of task-threads

Algorithm 5 Batch Marking

```
marking_stack[]
batch_mark_stack[]
function PUSH(ref)
    batch_mark_stack.add(ref)
    if (batch_mark_stack).is_full() then
        start_batch ← time_stamp_counter()
        i ← 0
        while (i < batch_mark_stack.size()) do
            if test_and_mark(batch_mark_stack[i]) then
                set_marked_by_self(i)
            end if
            i ← i + 1
        end while
        end_batch ← time_stamp_counter()
        record(end_batch - start_batch)
        i ← 0
        while i < batch_mark_stack.size() do
            if marked_by_self(i) then
                push_ref(marking_stack)
            end if
            i ← i + 1
        end while
        reset_batch_mark_stack()
    end if
end function
```

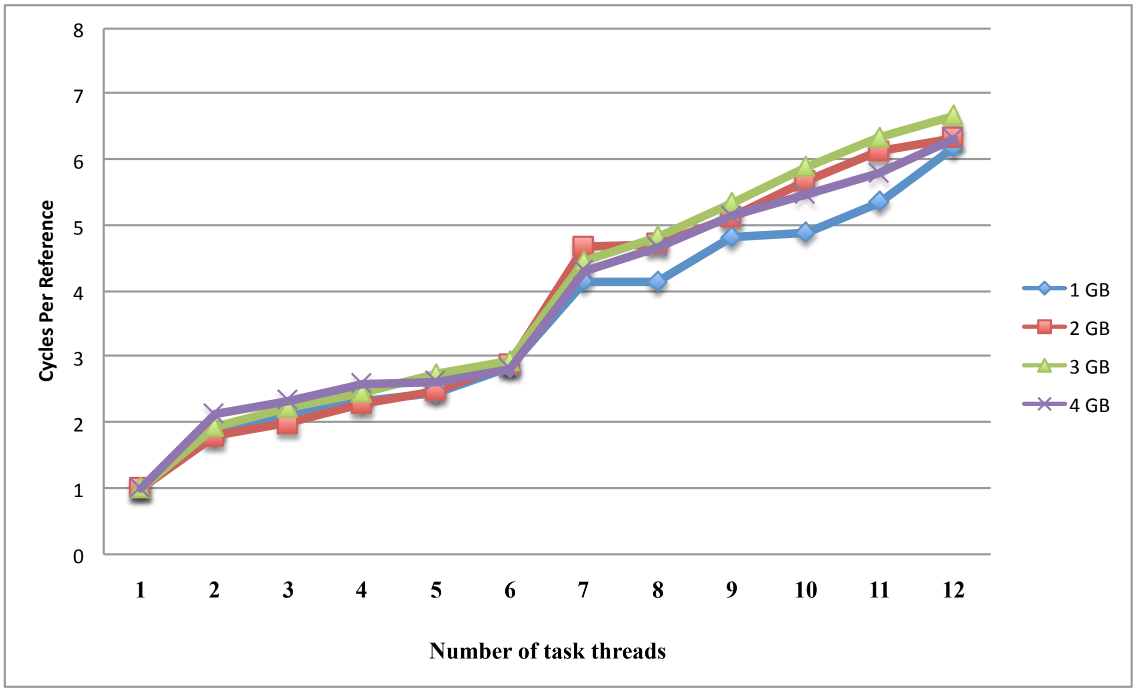

Figure 4.1: Cycles to mark a reference increases with the number of task-threads

for different heap sizes. The numbers are normalized by the cycles per reference for a single task-thread running in stop-the-world mode. As can be seen, cycles-per-reference gradually increases and follows a trend that is independent of the heap size. Fig 4.2 plots the speedup in wall-clock mark times relative to the number of task-threads for different heap sizes. The metric is normalized by the mark time for a single task-thread in stop-the-world mode. We continue to get a speedup by adding additional task-threads even though the mark-cycles-per-reference continues to creep up. This expected throughput increase is due to increased parallelism, although the single-threaded performance decreases. At some point in the graphs, the contention between the task-threads and the NUMA effects start to dominate, which results in the flattening of the speedup curve. On the Westmere-EP architecture there is a relatively big jump in mark-cycles-per-reference around 7 task-threads, which we think is because of NUMA effects on the 6 cores-per-socket machine.

Measuring Contention During Atomic Updates to Mark-Bitmap

Atomic update instructions, such as CAS, implement the atomicity semantic by providing mutual exclusion at the hardware level. On the X86 platform, the “lock” prefix, used in conjunction with the CAS instruction, results in a processor/core getting exclusive access to the system bus,

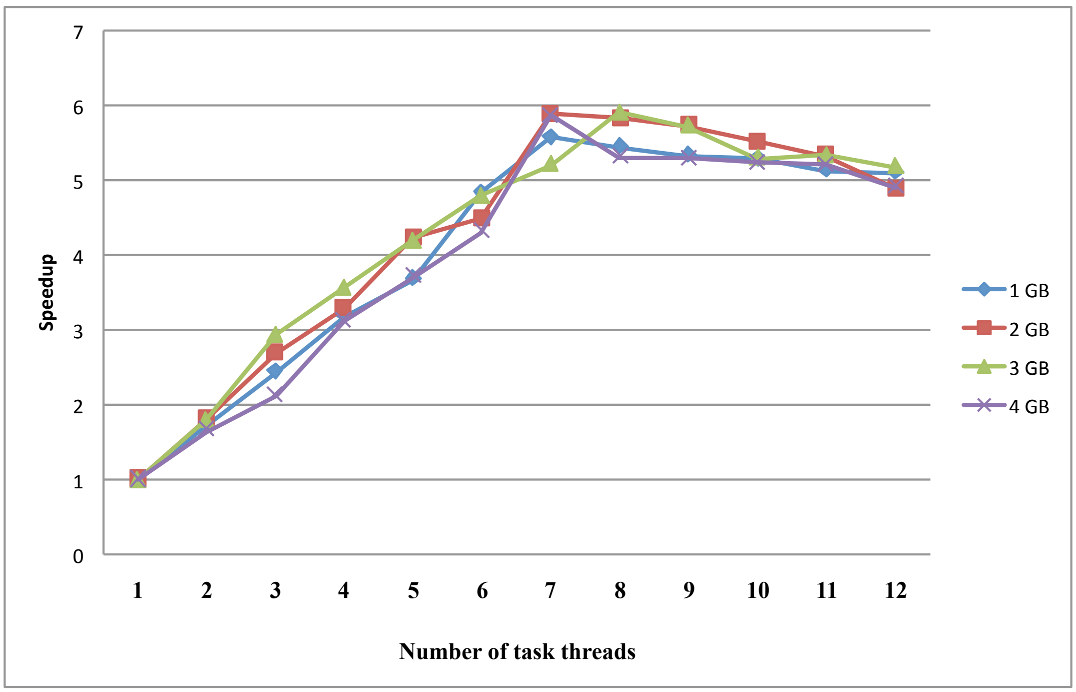

Figure 4.2: Speedup in parallel marking time flattens out as the number of task-threads crosses a threshold

thereby providing mutual exclusion. Newer X86 processor architectures don't lock the system bus if the memory word being updated is present in the processor cache and is completely contained within a cache line [1]. In these cases, the cache-coherency mechanism ensures atomicity. Other hardware platforms also benefit significantly from similar optimizations. Atomic updates of a memory word by different processors/cores can result in significant serialization of hardware resources and cache line ping-ponging.

In the case of parallel marking based on purely processor-centric work-sharing, our hypothesis is that a single word in the mark-bitmap is atomically updated by several task-threads resulting in contention that limits marking throughput. In order to verify this hypothesis, we track the number of unique task-threads that atomically update a set of words in the mark-bitmap that map to the same cache line. Our scheme assigns a unique thread-id to each task-thread and allocates a separate "mark-id" byte-map at the start. This byte-map is sized in a similar manner to the mark-bitmap and can store a byte value for every object in the heap. A task-thread that successfully marks an object "live", stores its thread-id in the byte corresponding to the object in the mark-id byte-map. We don't need an atomic store for the write in the byte-map, since only one thread can successfully mark an object. At the end of marking, we walk through the mark-bitmap, a cache line at a time. For each bit that is set in the cache line of the mark-bitmap, we look up, in the mark-id byte-map the number of unique thread-id's that atomically updated the cache line.

Fig 4.3 plots the number of unique task-threads that write to a single cache line in the mark-bitmap. As suspected, in our baseline processor-centric work-sharing scheme, this number increases as the number of task-threads increase. On a multi-processor system with multiple cores per processor, this pattern will result in significant cache traffic, inhibiting throughput.

4.4 Scalable Work Sharing at Cache Line Boundaries

Our core premise is that there is contention due to multiple GC threads attempting to atomically update words in the mark-bitmap that map to the same cache line. This contention is a limiting factor in the scalability of parallel marking and we aim to reduce this.

Our algorithm is based on distributing the marking work onto N stripes. Each object reference maps to a unique stripe. The stripe index for a reference is based on the address of the mark-bitmap word that corresponds to the reference. Similarly, each task-thread maps to a unique stripe, i.e., is responsible for draining work from a single stripe. However, we could have multiple task-threads mapping to the same stripe. In a system with a total of M task-threads, M/N task-threads are responsible for draining the work in each stripe. A task-thread's stripe index is determined by modulo arithmetic, i.e., *task-thread-id* % N. This scheme ensures that a maximum of M/N threads can update a single word in the mark-bitmap. In our baseline

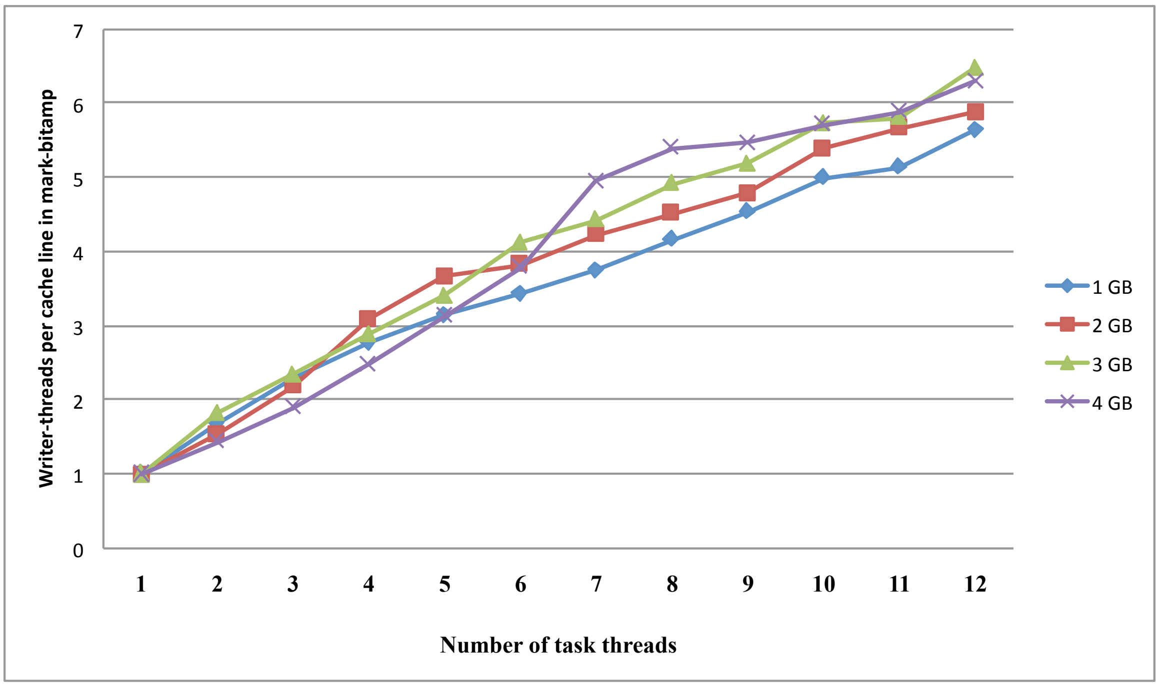

Figure 4.3: Number of task-threads writing to a cache line in the mark-bitmap relative to number of task-threads

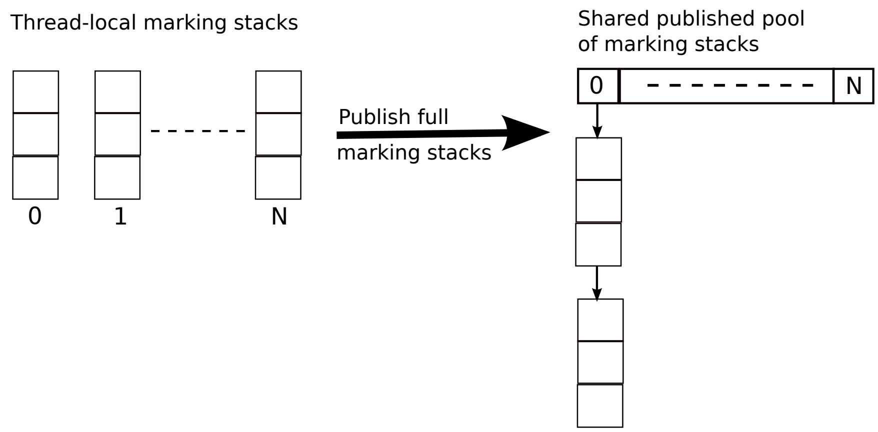

Figure 4.4: Marking data structures for a task thread

algorithm, $N = 1$, implying that all the task-threads could potentially update a single word in the mark-bitmap. On the other hand, when $N = M$ one task-thread can update a single word in the mark-bitmap.

4.4.1 Data Structures

In order to stripe the work N ways, every task-thread starts off with an array of N private thread-local marking stacks. Of these, the stack corresponding to the task-thread's stripe-index is treated as its current marking stack. The task-thread uses its current marking stack in the usual manner for "marking through" references that belong to its work-stripe. The other $N-1$ "work-sharing" marking stacks are used by the task-thread to generate marking work for other task-threads. Note that a task-thread both pushes to and pops from its current marking stack, while it only pushes work onto the other $N-1$ work-sharing stacks.

A task-thread's N marking stacks are private and invisible to the other threads, until they are "published". Each task-thread maintains a "published pool" of marking stacks, striped N ways. A full marking-stack is pushed on to the task-thread's published pool and at that point becomes eligible to be stolen by other task-threads. A full marking-stack is added to the same stripe-index in the published pool corresponding to its original stripe-index. Each stripe in the published pool is maintained as a singly linked-list of marking stacks. Pushing a marking-stack onto the published pool is a lock-free operation and can be achieved via a CAS instruction. Each task-thread also maintains a thread-local pool of free marking stacks in order to avoid the ABA [8] problem. Fig 4.4, illustrates the data structures for each task-thread.

4.4.2 Algorithm

We describe our algorithm in a format used by the Jones book [42], focusing on the key sub-tasks of acquiring, performing and generating "marking" work.

Acquire work: A task-thread starts off popping references from its current marking stack. As in the baseline algorithm, working on the current marking stack doesn't require any atomic operations. Once that is done, it attempts to acquire full marking stacks from its stripe in the published pools. The task-thread starts off looking for work in its own published pool and then moves on to the published pool of other task-threads. Each time it only acquires work from the work-stripe it is responsible for. Acquiring work from a stripe is a lock-free operation done using CAS operations.

Perform work: For every reference that the task-thread acquires, it enumerates the pointers to its child objects. For each non-null reference to a child object, the task-thread has to decide whether it falls into its work-stripe. This check has to be made prior to testing whether the object is already marked, in order to avoid cache pollution in the case where the object doesn't fall into the task-thread's work-stripe. The task-thread performs the usual test-and-mark operation on a reference that maps to its work-stripe. References that map to other work-stripes are pushed onto the correct thread-local work-sharing marking stack; the task-thread doesn't read or update the corresponding word in the mark-bitmap for these references.

Generate work: A task-thread adds full marking stacks to the right stripes in its published pool. Partially full marking stacks owned by a task-thread are not visible to other task-threads. This is necessary to avoid a race between the "owner" task-thread and the "stealer" thread, where the owner thread could still be pushing references onto an already stolen stack resulting in stack corruption. However, this can result in work being kept away from potentially idle task-threads. We reduce the possibility of this happening by having the task-thread flush the partially full private marking stacks to its published pool each time it finishes draining its current marking stack.

Termination condition: The task-threads spin on a single status word that is updated atomically. The bottom M bits of the status word, with a bit for each of the M participating task-threads, indicate availability of work for the corresponding task-thread. The top $\log_2 M$ bits of the status word are used to maintain a count of active task-threads. The two fields need to be part of a single word, since they need to updated atomically.

During tracing, each task-thread can potentially push work onto any other work-stripe, including the work-stripes that it is not responsible for. A task-thread needs a mechanism to signal to the other task-threads that there is new work available on their stripes. This is critical, since task-threads only drain work from the stripes that they map to. The bit-mask in the status word is used for this purpose. However, the bit-mask by itself is not enough to ensure that the work generated by one task-thread for another task-thread is actually picked up by the responsible task-thread. There is a time window where a sibling task-thread can exit its main work loop before it sees the bit set in the status word. In order to close out this race, we need a count of active task-threads to be part of the same status word.

At the start, all the bits of the bit-mask in the status word are set. This ensures that all task-threads will enter the main work loop at least once. Before it enters the main work-loop, a task-thread clears its bit in the bit-mask and increments the active thread count with a single atomic operation. By incrementing the active thread count in the status word, it ensures that even task-threads that don't currently have any work in their stripes continue to be in the work loop. As part of the tracing loop, each task-thread generates a bit-mask with a bit corresponding to each of the sibling task-threads that now have new work available for them. The task-thread does this by setting a bit in a local bit-mask, each time it pushes work on to a stripe in the published pool. Once the task-thread exits its tracing loop, it atomically updates the status word with a new value. The new value of the status word has a decremented active thread count and a bit-mask that is obtained by or'ing the current bit-mask of the status word with the bit-mask generated by the task-thread. This ensures that task-threads will continue to drain newly generated work in their stripes even if the active thread count momentarily falls to zero. Note that while a task-thread can set bits in the bit-mask for other task-threads, these bits can only be cleared by its "owner" task-thread.

Marking concludes when every task-thread has drained its current marking stack, doesn't find any work in its stripe in any of the published pools and generates no new work. The pseudo code for the proposed algorithm is outlined in Algorithm 6

4.4.3 Discussion

We now discuss some of our design choices. Having presented our algorithm, this is also a good opportunity for us to explain how our approach relates to a couple other algorithms that share certain properties with ours.

Load balancing versus synchronization costs: We rely on the proposed work-sharing algorithm to reduce contention during atomic updates to the mark-bitmap as well as for load-balancing. The algorithm tries to reduce synchronization costs by handing work in chunks. Per thread published pools instead of a global published pool also serve to reduce contention while publishing new work.

Stack sizes: The size of the marking stacks plays an important role in the performance of the algorithm. Large work-sharing marking stacks can result in a task-thread generating work that remains unavailable to other threads, since this work is mainly published when these stacks get full. Work-sharing marking stacks with very few entries, can result in frequent pushes to the

Algorithm 6 Work Sharing at Cache-Line Boundaries

```
shared published_pool[M][N]                              ▷ M task-threads and N stripes
shared status_word
private marking_stack[N-1]              ▷ work for other task-threads is pushed onto these
private current_marking_stack
me = myThreadId
my_stripe = me
function ACQUIRE_FROM_PUBLISHED_POOL((current_marking_stack))
    for task_thread_id = 0 → M do
        for stripe = 0 → N do
            if (stripe == my_stripe) then
                current_marking_stack=get(published_pool[task_thread_id][stripe])
            end if
        end for
    end for
end function
function DRAIN_MARKING_STACK()
    while !current_marking_stack.is_empty() do
        ref ← pop(marking_stack[me])
        bitmask| = mark_through(ref)
    end while
end function
function MARK_THROUGH(ref)
    child_pointer_list = ref.child_pointer_list()
    while child_pointer_list do
        child = p.load()
        if (child! = NULL) then
            if (my_stripe == stripe_for_reference(child)) then
                if child.test_and_mark()) then
                    bitmask| = current_marking_stack.push(child)
                else
                    bitmask| = marking_stack[child.stripe()].push(child)
                end if
            end if
        end if
    end while
    return bitmask
end function
```

Algorithm 6 Work Sharing at Cache-Line Boundaries (continued)

```
function STRIPE_FOR_REFERENCE(child)
    return (mark_word(child)%BytesPerCacheLine)%N
end function
function PUSH(ref )
    add(ref)
    if self.is_full() then
        bitmask ← add_to_published_pool(self, stripe_for_reference(ref))
    end if
end function
function ADD_TO_PUBLISHED_POOL(stack, stripe)
    published_pool[me][stripe].add(stack)
    return bitmask ← (1 << stripe)
end function
function MARK_TASK()
    while status_word > 0 do
        inc_working_count_clear_bit(me)
        bitmask ← drain_marking_stack()
        dec_working_count_set_bit_mask(bitmask)
    end while
end function
```

published pool, thereby causing contention. Also, a task-thread's current marking stack should be large enough for it to continue working without frequent overflows. We find that asymmetrical stack sizes with a large current marking stack and small work-sharing marking stacks perform best. This allows a task-thread to continue on its share of the work while frequently publishing work for other task-threads.

Number of work-stripes: Our primary goal is to reduce contention during updates to the mark-bitmap. A 1:1 mapping between the task-threads and the number of stripes performs best in this regard. A 1:1 mapping implies that each task-thread have $M - 1$ work-sharing marking stacks. In most architectures, this results in a situation where the marking stacks of an individual task-thread quickly occupy the L1 cache. Performance degrades in this case, since we have just traded coherence misses for capacity misses. In practice we find that a 1:1 mapping is not necessary and a 1:4 mapping effectively filters out bulk of the contention.

Edge enqueuing: This is a property of a marker where all non-null object references are pushed onto a marking stack without a test-and-mark operation. Our algorithm shares this property with the algorithm proposed by Garner et al. [32]. However, while their primary motivation is to improve the single-threaded performance by increasing the effectiveness of prefetching, our goal is to increase scalability in a parallel framework by reducing contention. Garner et al. [32], show that edge-enqueuing helps the most in a single-threaded environment when the mark-bit is located in the object header. Since our main premise is the use of a side mark-bitmap, we claim that our performance benefits are not purely because of edge-enqueuing but rather due to reduced contention during atomic updates to mark-bitmaps. As observed by Garner et al. [32], edge-enqueuing has the side-effect of pushing an increased number of object references onto the marking stacks. We concur with their observation that these extra, thread-local pushes/pops don't seem to affect performance negatively.

Single-writer, single-reader channels: The work-sharing algorithm proposed by Wu et al. [68] exhibits some similarities to our algorithm. In their scheme, each task-thread is required to have an array of $M - 1$ work-sharing queues, where M is the total number of task-threads. There is always a 1:1 mapping between the number of per-thread work-sharing queues and the number of threads, which allows them to avoid atomic operations during load-balancing. They avoid capacity misses by picking very small queues; they get their best performance with queue sizes of 1 or 2. Our algorithm is not restricted to a 1:1 mapping between task-threads and the number of per-thread work-sharing stacks and hence can support larger individual work-sharing stacks. Wu et al. [68] also indicate that the scalability of their algorithm suffers when they use atomic operations to update the mark-bit, which in their implementation, is located in the object header. Having the mark-bit in a side mark-bitmap would only make the situation worse. This gives further credence to our hypothesis that atomic updates to the mark-bitmap are a fundamental source of bottlenecks in parallel marking algorithms.

4.5 Results

We now evaluate the effectiveness of our work-sharing scheme using the same metrics as before. Our work-sharing algorithm relies on striping the work N ways with M/N threads mapping to each stripe, where M is the total number of task-threads. Each task-thread also starts off with $N-1$ work-sharing marking stacks. As stated earlier, this mapping ratio affects the performance of our algorithm in a couple different ways. A smaller mapping ratio implies less contention between the task-threads during the mark-bitmap updates. However, it also implies a larger number of per-thread work-sharing marking stacks, which can result both in memory overhead as well as capacity misses. We vary this critical ratio to measure its effect on the performance of the proposed algorithm. We tweak the sizes of the marking stacks and the plots reflect the best sizes for the set of benchmarks. The marking stacks in our baseline algorithm have 2K entries each, sized to keep the memory overhead low. In our runs, a current marking stack with 1K entries and work-sharing stacks with 128 entries each give us our best performance numbers. As seen in the performance numbers of our baseline algorithm, the heap-sizes don't have an effect

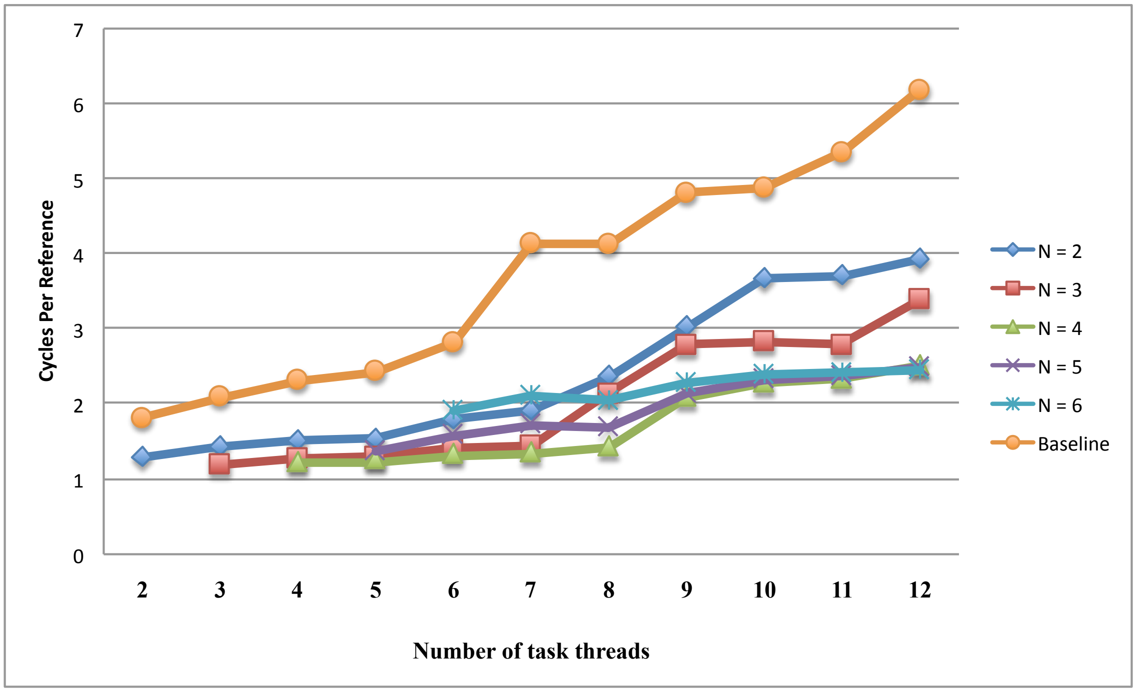

Figure 4.5: Cycles to mark a reference relative to number of task-threads scales much better compared to the baseline algorithm

on our metrics. We find this to be true for our proposed algorithm as well; the plots reflect numbers with a heap size of 1GB.

The proposed algorithm requires that there are atleast as many task-threads as there are work-sharing stripes. This is because threads drain work only from their assigned stripes; hence a lesser number of task-threads than the number of stripes would result in stripes that wouldn't be drained. Our plots reflect this property of the algorithm. In each of our plots, we also add the corresponding measurement for the baseline algorithm. All the metrics are normalized by the corresponding metric for the single-threaded stop-the-world mode.

Fig 4.5 plots the mark-cycles per reference relative to the number of task-threads for different number of work-sharing stripes. The value creeps up as the number of task-threads increases but stays well below the baseline measurement. As would be expected, we see a reduction in the mark-cycles-per-reference as the M/N ratio decreases with an increasing number of work-sharing stripes. Fig 4.6 plots the speedup in wall-clock mark times for the proposed algorithm. The baseline algorithm starts to flatten out and has a drop in performance beyond 7 to 8 task-threads; the proposed algorithm continues to scale albeit gradually. The proposed algorithm sees a 15-20% increase in speedup when the task-thread count goes over 4. For smaller counts

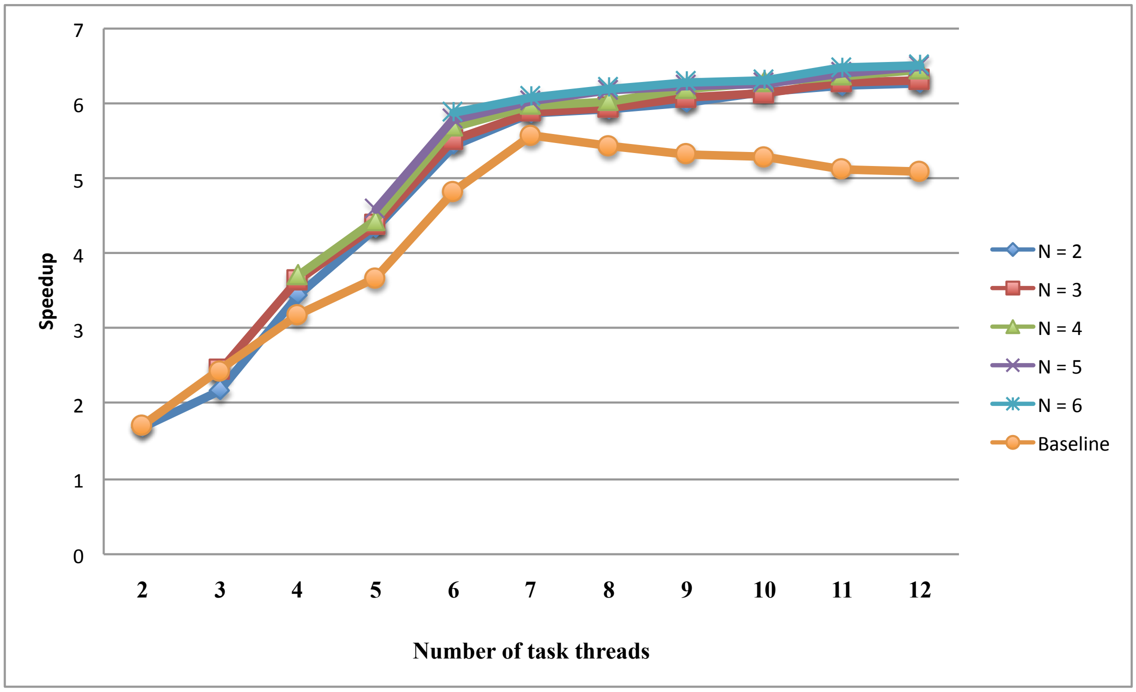

Figure 4.6: Speedup in parallel, marking time continues to scale with the number of task-threads

of task-threads, the performance is very close to the baseline algorithm. This is because, with smaller counts of task-threads, the baseline algorithm suffers less from contention as expected, while the proposed algorithm continues to pay the additional work-sharing costs, i.e., that of enqueuing work into the work-sharing marking stacks and publishing these stacks. Note also that a cache line in the mark-bitmap is a good granularity for work-sharing.

Fig 4.7 demonstrates the effectiveness of our striping mechanism in reducing the number of task-threads that update a single cache line in the mark-bitmap. This metric stays well below the total number of task-threads per work-sharing stripe. This implies that our scheme is an effective means of reducing contention even without a 1:1 mapping between the task-threads and the work-sharing stripes.

4.6 System-Topology-Aware Work Distribution

Modern SMP systems generally have multiple multi-core processors. The processor topology defines the number of physical packages, the number of cores in each package and the number of hyperthreads per core. Processor cores that are part of the same physical package are referred

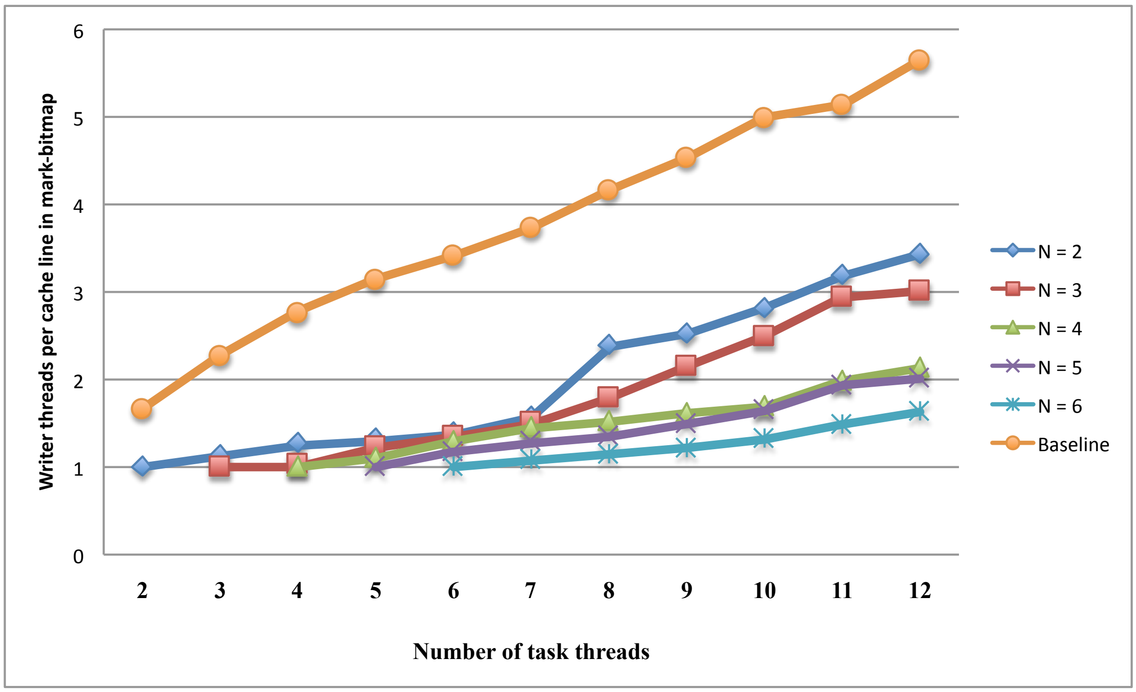

Figure 4.7: Number of task-threads writing to a cache line in the mark-bitmap relative to number of task-threads

to as sibling cores. Hardware execution threads on a hyper-threaded core are referred to as sibling threads. The cache topology defines the number of caches per core, the type of cache, e.g., instruction cache/data cache, the cache levels and the cache sharing among the sibling cores/threads. On Linux systems, this information can be obtained from the "sys" pseudo-filesystem (sysfs). As previously noted our Westmere-EP system has 2 sockets with 6 hyper-threaded cores per socket. Each of the 6 cores in the same socket have individual L1/L2 caches, but they do share the L3. The hyper-threads on each core share both the L1 and the L2 cache. The L2 is non-inclusive while the L3 is an inclusive cache. This is a fairly common cache topology on current systems.

Most modern multiprocessor systems, such as our Westmere-EP platform, are cc-NUMA, shared-memory platforms. On cc-NUMA machines, each node is assigned some amount of local memory. The cost of accessing remote memory is significantly higher than the cost of accessing local memory. On Linux systems, the "sys" pseudo-filesystem also publishes the "NUMA" information for the system, including the amount of memory per node, the memory layout, and the cost of memory accesses at each node[2]. We consider the processor and the cache topology along with the per-node NUMA information to be the "system topology." The system topology, along with the cost of accessing the different levels of caches and the memory at different nodes, paints a full picture of its effect on the performance of a multi-threaded application.

Prior work on NUMA-aware garbage collection algorithms has primarily been in the domain of copying collectors, with the focus being on improving mutator performance by copying objects to nodes that are most likely to access them [51, 64]. Our scheme is the first of its kind to make use of the system topology information to increase marking performance.

Concurrent and on-the-fly collectors [63, 27, 9] have to deal with mutators accessing references to potentially "white" objects and propagating these references into "black" objects while marking is still in progress. These algorithms generally have mutators collect these references in thread-local buffers using barrier mechanisms. These buffers are handed over to the marker to be marked through via synchronization mechanisms such as checkpoints [63, 9] or at stop-the-world safepoints. A "trap-on-access" mechanism, along with general locality principles, would imply that there is a high likelihood of these objects continuing to be in the cache of the processor/core that executed the mutator trap code. A collector task-thread that is scheduled on a sibling core/thread of the core that executed the mutator trap code could then potentially benefit, if it were assigned the work of marking through these objects. This is similar to techniques that use hyper-threads as helper threads for prefetching purposes [28]

In our implementation of the baseline C4 algorithm, each mutator thread gets a thread-local marking stack. During the concurrent mark phase, mutators that trigger the "lvb" style read-

[2]The processor and cache topology information can be obtained from: /sys/devices/system/cpu/cpu*/cache The NUMA information can be obtained from: /sys/devices/system/node

barrier [63] on access to a potential "gray" reference, record it in their thread-local marking stack. These mutator marking stacks are pushed onto a global pool of mutator marking stacks at a checkpoint [63]. The collector's task-threads acquire these marking stacks from the "mutator marking stack pool" and mark-through the references in these stacks.

In our topology-aware work-distribution scheme, we maintain the mutator marking-stack pool as an array of head pointers to linked lists of mutator marking stacks. The processor-id serves as the index into this array. A mutator attempting to add its marking stack to this pool, looks up its current processor-id using the CPUID [1] instruction and uses that as the stripe-index. It is possible that the mutator has queued most of the work in its marking stack while running on a different core. However, we find this to be rare, since the scheduler generally tends to schedule threads on the same core as before.

When a collector task-thread looks for work in the mutator marking-stack pool, it looks up its processor-id and prioritizes indexes in the pool that map to its sibling threads and sibling cores. We therefore refer to this scheme as "processor-affinity" based work-distribution. This is only an optimization, and all stripes in the mutator marking-stack pool are eventually drained. However, this does serve as a very effective filter. The processor topology map is read once at the beginning using the "sysfs" pseudo file system.

4.6.1 Evaluation

While working through the references in a mutator marking stack, the collector's task-threads would likely benefit from a processor-affinity based policy by finding the objects being marked through in the processor cache. In order to evaluate this algorithm, we turn off prefetching. We also turn off the mark-bitmap address-based striping. We use the previously discussed evaluation method for batch marking to measure the average number of processor cycles to mark through an object . We now include the enumeration of child pointers as part of the batch. We only collect our metrics when the task-threads work on the mutator marking stacks in the concurrent mark mode. To evaluate this algorithm, we use the Volano 2.9.0 [4] benchmark, a highly multi-thread benchmark that measures the performance of the Volano chat server. A highly multi-threaded benchmark like Volano is more likely to have a larger number of mutator threads that generate marking work, and is likely an ideal benchmark for our proposed algorithm.

Fig 4.8 plots the relative speedup in the processor cycles for marking-through a reference, normalized by the single-threaded measurement in the concurrent mark mode. As expected, for smaller counts of task-threads we don't see much benefits since task-threads would be expected to drain work from all the processor stripes. We see the biggest jump in speedup when we go to 7 task-threads, which is likely because of NUMA effects on our 6 processor per socket Westmere-EP machine. We see a performance benefit of about 11% for the maximum alloted

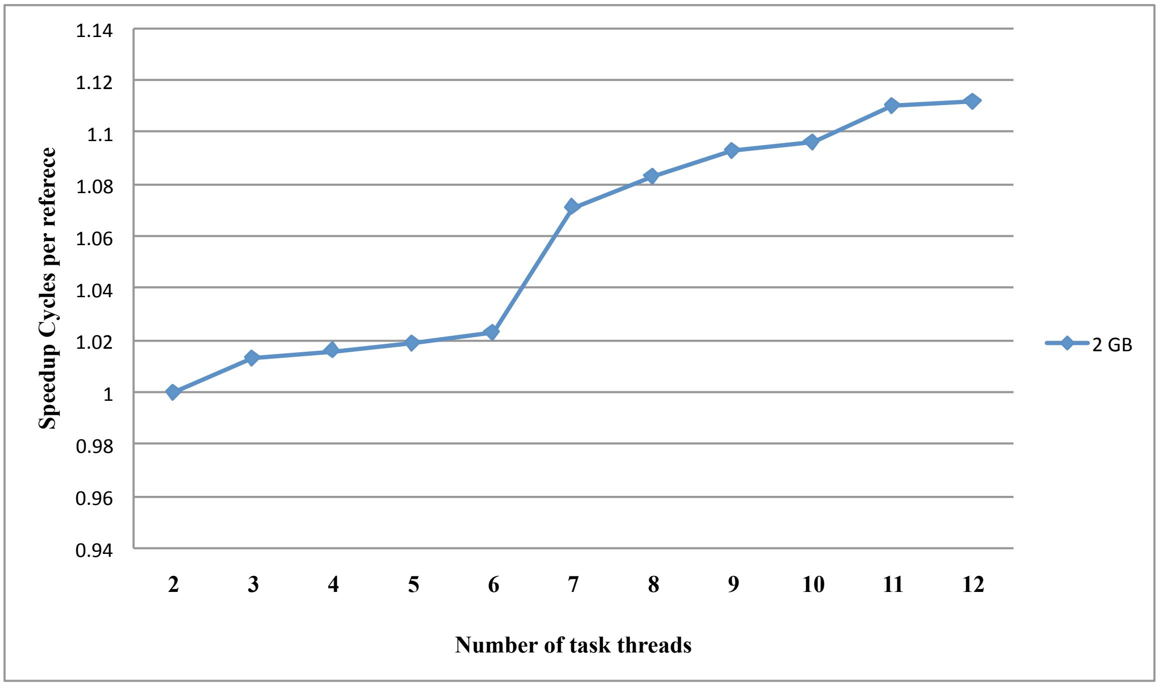

Figure 4.8: Cycles per reference marked through relative to number of task-threads

threads per generation. We restrict ourselves to a maximum of 12 task-threads per generation on our 24-core machine to account for the inter-generation concurrency of the C4 [63] collector that can result in both generations being garbage collected at the same time. We use the default scheduler mechanism and don't use any scheduler affinity calls to divide the task-threads among the 2 sockets.

4.7 Conclusion

We have studied the performance and scalability of a concurrent, parallel marking algorithm that uses a mark-bitmap to maintain object liveness information. Our evaluation methodology allows us to tease out the test-and-mark operation and measure its performance and scalability. We also shed light on the contention between the collector's task-threads while atomically updating the mark-bitmap. We use these to make the case that in a modern SMP system a purely processor-centric work-sharing algorithm has inherent scalability limitations. We then describe our new non-blocking and lock-free work-sharing algorithm, which distributes work based on the addresses in the mark-bitmap. It is a first of its kind, "hybrid" work-sharing algorithm that uses the memory address of a word in the mark-bitmap, to both reduce contention among the task-threads as well as keep them busy. We observe a 20% improvement in parallel marking times with this scheme. We finally describe our processor and cache topology aware work-distribution algorithm for mutator-generated marking work applicable to parallel, concurrent/on-the-fly markers. We believe that the topology-aware algorithms should be a focus of the research community, considering the fact that cc-NUMA architectures are quite common place in modern servers.

Chapter 5

Concurrency and Parallelism in the Virtual Machine

In this dissertation, we have discussed three distinct algorithms in this dissertation pertaining to garbage collected runtime environments. Chapters 2 and 3 described two concurrent compacting garbage collection algorithms, both aimed at improving the consistency of application response times and application throughput, by providing low-cost and wait-free [37] read-barriers. We begin this chapter with a short summary of the motivation for each of these algorithms, the design goals, the algorithms and their performance characteristics. We then discuss another use of transactional memory within the virtual machine, primarily its use in speculative lock elision. [54] We provide some context for this discussion by first describing the implementation of language-level synchronization constructs, i.e., locks, within the virtual machine. We end the discussion with a note on the usage of the transactional-memory hardware that is going to be omnipresent in upcoming server systems. [3]

Chapter 4 of this dissertation described a new work-sharing algorithm between parallel task-threads used for marking the application heap. We summarize our work-sharing algorithm here and delve into some of the recent work-sharing algorithms described in literature.

5.1 Concurrent compaction

Garbage-collection algorithms deal with the issue of memory fragmentation by compacting the application live-set into contiguous memory locations. As noted in earlier chapters, compaction involves moving a live object and updating the pointers to its old location. Compacting the heap while the application threads are active, i.e., concurrent compaction, poses a special set of challenges that deal with synchronization between the mutator threads and the garbage collector. Historically, industrial implementations of compacting collectors have sidestepped the issue of concurrent compaction by splitting the heap into a “small” new generation and a relatively large old generation. These implementations rely on frequent, stop-the-world compaction of the “small” new generation in order to sustain the application allocation rates. These implementations do have to pay the penalty of the occasional, stop-the-world compaction of the old generation.

However, modern multi-core architectures, growing memory bandwidths and availability of cheap, abundant memory are pushing the envelope in terms of application allocation rates and the size of the new generation. It is not uncommon to see allocation rates in the 10s of GB/sec in multi-threaded, application servers and middleware applications that run on modern multi-core architectures with large memory bandwidths. This easily manifests in a few gigabytes of live memory in the new generation. A stop-the-world compaction of a new generation of this size will easily result in application pauses that are larger than 100s of milliseconds. At the same time application profiles have continued to change dramatically; securities trading applications and web-tier applications often have response-time requirements in the range of a few milliseconds, and a GC induced application pause that lasts a few 100 milliseconds is unacceptable. The inability of current implementations of compacting collectors to sustain high application allocation rates along with changing application profiles with ever shrinking response time requirements, has fuelled the development of a new family of concurrent compacting collectors in current JVMs. [21, 63, 44]

Having made a case for concurrent compacting algorithms, we confront the primary bottleneck in the design of these algorithms: the cost of synchronization between the mutator and the concurrent collector. Read-barriers in the application code are used for synchronizing between a concurrent compacting collector attempting to relocate an object and a mutator trying to access it at the same time. Most read-barrier implementations cooperate with the concurrent collector. They help out by doing some of the GC related work if they access objects that are marked for relocation but have not yet been relocated. This results in mutators spending a significant amount of their time in executing GC-related code rather than application code, causing a drop in throughput and jitters in application response.

We present a short summary of the two collectors described in chapters 2 and 3 that use copy-aborting read-barriers that significantly reduce the mutator synchronization overhead.

5.1.1 The Collie and the Single-Stateword Collectors

The Collie collector builds and maintains a per-object stable referrer set, i.e., a set of pointers to an object. Object transplantation involves both copying the object to its new location and updating the elements of its referrer set. The key idea is that mutators can access an object only if they have access to one of the elements of its referrer set and hence it is necessary to update the referrer set in a thread-safe and consistent manner. The Collie GC-protects the objects it intends to transplant, thereby ensuring that read-barriers will trigger if mutators attempt to access any of these objects. Read-barriers that trigger prior to an object being relocated don't race with the collector and are able to abort the object copy. The Collie uses hardware transactional memory to update the elements of an object's referrer set. A read-barrier that triggers with the collector in the middle of a transaction to update the object's referrer set will write to an element of the transaction's write-set, thereby ensuring that the transaction fails and the object copy is aborted. The Collie use of transactional memory allows it to detect conflicts directly when any element of the referrer set is written to. The resulting read-barrier is a constant-cost, wait-free barrier that comprises a single CAS and a load instruction.

The motivation for the single-stateword collector was to implement the Collie's wait-free read-barrier without the use of transactional memory. The collector distills a heap object's multi-word transplantation state into a single word comprising its "GC-state" and a forwarding pointer. The collector and the read-barrier code synchronize on the single stateword using an atomic update instruction that is widely available on most platforms: a CAS instruction. Both the collector and the read-barrier try to claim the rights to transplant the object by CAS-updating the stateword, with the winner then proceeding to transplant the object either to the mirrored-to-space or the new to-space location. The read-barrier continues to be wait-free albeit slightly more expensive than the Collie's read-barrier.

Both collectors are able to provide significant MMU improvements even in the sub-5ms time windows compared to the Pauseless collector [21] on the Vega architecture. They also match it in its ability to scale to large heap sizes and provide high throughputs. As expected, the Collie provides better response times and throughputs compared to the single stateword collector.

5.2 Implementation of Language Level Synchronization in the Virtual Machine

We now discuss the use of transactional memory to speedup language-level synchronization and contrast it with the Collie's use of transactional memory. We provide some context by discussing the JVM's implementation of language-level synchronization constructs.

Language-level synchronization constructs, such as the "synchronized" keyword in Java, need to provide the semantics of a lock to the application code. They need to provide mutual exclusion and provide wait/wake-up semantics. JVMs generally use OS-level locks to support these constructs. However, acquiring and releasing these OS-level locks has a high overhead. Over the past two decades, JVMs have implemented several optimization techniques to reduce some of this overhead. We list a couple of these below, before discussing the use of transactional memory in this context.

5.2.1 Thin Locks

Lock-based, algorithms are hard to get right, and they are often the primary bottlenecks to application scalability. The usage pattern of locks in Java applications indicates that a significant percentage of locks are never contended. In these cases, the locks don't need to acquire/release an OS-level lock. Thin locks allow the application thread to take advantage of this fact, allowing it to "acquire" the object lock by atomically updating a word, viz., the owner-id field with its thread-id. Lock-release is generally a simple store instruction that stores a $NULL$ in the owner-id field. The JVM can detect contention on thin locks and transition them seamlessly to OS-level locks. However, this "inflation" of thin to thick locks is rare and thin locks deliver a significant performance boost.

5.2.2 Biased Locking

The CAS operations used for thin locks incur considerable local latency in the executing CPU. It turns out that most objects are locked by at most one thread during their lifetime. Biased locking allows that thread to "bias" an object lock toward itself. Once biased, that thread can subsequently lock and unlock the object without resorting to expensive atomic instructions. An object can be biased toward at most one thread at any given time. If another thread tries to acquire a biased object, however, the JVM needs to detect this and seamlessly revoke the bias from the original thread. An equivalent way to conceptualize biased locking is that the original owner simply defers unlocking the object until it encounters contention. Revoking a biased lock is a fairly involved process and is beyond the scope of this dissertation. However, it is important to note that for biased locking to be profitable, the benefit obtained by eliding compare-and-swap instructions should exceed the cost of lock revocation. Biased locking is a response to the latency of the CAS instruction. The benefits of biased locking have been diminishing in the past few years as hardware optimizations continue to decrease the latency of the CAS operation.

5.2.3 Transactional Memory use in the Virtual Machine: Speculative Lock Elision

We will now look at how a JVM[1] can use transactional memory to provide optimistic concurrency in the case of contended locks. The observation here is that contended locks can often have disjoint data-sets despite the fact that the locks are truly contended, i.e., multiple threads are attempting to obtain the lock at the same time. This happens primarily because application programmers often use coarse locking schemes where a single global lock could be used to "protect" a fairly big data structure. For example, application threads attempting to write to separate sections of a large hash table will end up contending for this global lock although they might be writing to different slots in the hash table. This causes unnecessarily throttling of throughput. The JVM can make use of hardware transactional memory in these cases by speculating that threads attempting to acquire the lock have disjoint data-sets and eliding the lock acquire/release. The transaction fails if this assumption doesn't hold and the threads fall back to a conventional OS-level lock for mutual exclusion. The benefits of the JVM's transparent use of hardware transactional memory to speed up language-level synchronization are:

1. No need for any changes to language specification.

2. No need to change application/library code: old code runs faster.

However, speculative lock elision within the JVM doesn't seem to buy as much performance as one would expect. The key observation is that most of the time speculative lock elision fails because of conflicts rather than capacity. This is mainly because application programmers and library writers don't write transactional-memory-friendly algorithms. Often these implementations update a shared counter of some sort within the transaction boundaries resulting in a conflict and transaction failure. A small rewrite of these algorithms will often result in successful transactions, however that breaks the "speed up the dusty-deck" goal that a lot of programs have.

In contrast, the Collie is built ground up relying on a transactional memory implementation. The algorithm makes deliberate design choices to ensure that a majority of the transactions succeed. The Collie avoids relocating objects with large referrer sets. Restricting the size of a transaction's write-set ensures that the transaction's write-set fits in the processor cache avoiding transaction failures caused by a transaction's write-set size exceeding the cache-size. Objects with large referrer sets also tend to be "popular" objects. Not relocating them allows the Collie to avoid transactions that have a higher failure probability due to conflicts with the application. The Collie refrains from relocating the application working-set, since these transactions would also have a much higher chance of conflict induced failure. These design choices allow a majority of the Collie's transactions to succeed. This highlights the fact that an algorithm designed and implemented with transactional-memory primitives as a cornerstone can make better use of this new hardware feature, as compared to schemes that rely on transparent replacement of locks in application code with transactions.

5.3 Work-sharing Among Parallel Marking Threads

The second part of this dissertation focussed on improving the throughput of parallel task-threads during the mark-phase of a garbage-collection algorithm. The cost of updating a word

[1] This derives from Azul's experience in implementing such a scheme on the Azul Vega platform

in the marking-bitmap via CAS operations was shown to be the primary source of bottleneck in scaling parallel marking speedup. This is mainly because multiple objects map to the same word in the mark-bitmap, and existing work-sharing algorithms aim to keep all the task-threads busy without any concern for contention among them. This results in lot of cache-coherency traffic and parallel mark speeds not scaling with the number of threads.

The non-blocking and lock-free work-sharing algorithm described in chapter 4 uses the address of a word in the mark-bitmap as the key to stripe work among parallel task-threads, with only a subset of task-threads working on each stripe. This filters out most of the contention during parallel marking. We also describe a practical way to measure the average number of processor cycles used for marking an object on the X86 platform using the platform's tick-counter register [1].

5.3.1 Work-stealing Algorithms

We now briefly discuss our experience implementing some of the other common work-stealing algorithms in the literature.

Steal-One Algorithm: Arora et al. [8] first described the non-blocking [37] steal-one algorithm where shared deques are used to hold and share pieces of work. The push onto the deque doesn't need any synchronization, while the pop needs synchronization only to claim the last element of the deque. The stealing thread always needs to synchronize. Synchronization happens via a CAS operation. The steal-one algorithm suffers from the ABA [49] problem, where a stealing thread can be tricked into believing that the state of the deque has remained unchanged although it transitions to a new value B and back to the value A that the stealing thread originally read. The steal-one algorithm gets around this by using the top bits of the deque's "top" field for a "tag" field. The tag field is incremented on every successful steal attempt. This ensures that a deque state is always unique barring a wrap-around in tag values. Flood et al. [31] use this algorithm for load-balancing in their parallel marking algorithm.

Steal-half Algorithm: The steal-half [36] algorithm is a generalization of the steal-one [8] algorithm that allows stealer threads to steal as much as half of a thread's work queue. This allows threads to steal more work for the same number of CAS operations and can potentially result in better load balancing. The steal-half algorithm also suffers from the ABA problem and deals with it in a similar manner to the steal-one algorithm.

In the context of parallel marking algorithms, our experience has been that the steal-half does no better than the steal-one algorithm in load-balancing and improving overall parallel mark times. We believe this is mainly because both of these "processor-centric" algorithms suffer from contention while updating the mark-bitmap via CAS operations. This fundamentally limits their scalability.

5.4 Conclusion

We have described three non-blocking algorithms in this dissertation, all aimed at improving the performance of garbage-collected environments. The Collie and the single-stateword collectors are both concurrent compacting collectors that aim to significantly reduce the application-visible synchronization overheads in runtime environments. We believe that such algorithms will be critical as applications consume memory at an every increasing pace, while requiring ever-shrinking response times. Finally, we described a non-blocking and lock-free work-sharing algorithm that allows for load-balancing while reducing the contention between task-threads carrying out the marking phase of a garbage-collection algorithm. As the number of cores in a system increases and NUMA machines become ever more prevalent, scalable "memory-centric" algorithms such as these will become vital.

REFERENCES

[1] Intel 64 and ia-32 architectures developer's manual: Combined volumes. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`.

[2] Intel 64 architecture processor topology enumeration. `http://software.intel.com/en-us/articles/intel-64-architecture-processor-topology-enumeration/`.

[3] Intel architecture instruction set extensions programming reference. `http://software.intel.com/en-us/avx/`.

[4] The volano benchmark. `http://www.volano.com/benchmarks.html`.

[5] Ole Agesen. Gc points in a threaded environment. Technical report, Mountain View, CA, USA, 1998.

[6] AMD Corp. Advanced Synchronization Facility. `http://developer.amd.com/tools/ASF/Pages/default.aspx`, 2009.

[7] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation*, PLDI '88, pages 11–20, New York, NY, USA, 1988. ACM.

[8] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA*, pages 119–129, 1998.

[9] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. pages 269–281.

[10] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 285–298, New York, NY, USA, 2003. ACM.

[11] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, New York, NY, USA, 2003. ACM.

[12] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 285–298, New York, NY, USA, 2003. ACM.

[13] Henry G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21:280–294, April 1978.

[14] Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Trans. Program. Lang. Syst.*, 27(6):1097–1146, 2005.

[15] 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*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[16] Stephen M. Blackburn and Antony L. Hosking. Barriers: friend or foe? In *Proceedings of the 4th international symposium on Memory management*, ISMM '04, pages 143–151, New York, NY, USA, 2004. ACM.

[17] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 157–164, New York, NY, USA, 1991. ACM.

[18] Hans-Juergen Boehm. Reducing garbage collector cache misses. pages 59–64.

[19] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 256–262, New York, NY, USA, 1984. ACM.

[20] Chen-Yong Cher, Antony L. Hosking, and T.N. Vijaykumar. Software prefetching for mark-sweep garbage collection: Hardware analysis and software redesign. pages 199–210.

[21] Cliff Click, Gil Tene, and Michael 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.

[22] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 336–346, New York, NY, USA, 2006. ACM.

[23] David Detlefs, Christine Flood, Steve Heller, and Tony 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.

[24] David Detlefs and Tony Printezis. A Generational Mostly-concurrent Garbage Collector. Technical report, Mountain View, CA, USA, 2000.

[25] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 157–168, New York, NY, USA, 2009. ACM.

[26] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Language Hierarchies and Interfaces: International Summer School*, volume 46 of *LNCS*, pages 43–56. SV, Marktoberdorf, Germany, 1976.

[27] Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for java. In *PLDI*, pages 274–284, 2000.

[28] Ulrich Drepper. What every programmer should know about memory. `http://www.akkadia.org/drepper/cpumemory.pdf`.

[29] Toshio Endo and Kenjiro Taura. Reducing pause time of conservative collectors. In *MSP/ISMM*, pages 119–131, 2002.

[30] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. Predicting scalability of parallel garbage collectors on shared memory multiprocessors. In *IPDPS*, page 43, 2001.

[31] Christine H. Flood, David Detlefs, Nir Shavit, and Xiolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[32] Robin Garner, Stephen M. Blackburn, and Daniel Frampton. Effective prefetch for mark-sweep garbage collection. In *ISMM*, pages 43–54, 2007.

[33] Alexander T. Garthwaite. *Making the trains run on time.* PhD thesis, Philadelphia, PA, USA, 2005. AAI3179735.

[34] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.

[35] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[36] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 280–289, New York, NY, USA, 2002. ACM.

[37] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149, January 1991.

[38] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[39] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[40] Richard L. Hudson and J. Eliot B. Moss. Sapphire: copying gc without stopping the world. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, JGI '01, pages 48–57, New York, NY, USA, 2001. ACM.

[41] JCP. JSR 166: Concurrency Utilities. `http://jcp.org/en/jsr/detail?id=166`, 2010.

[42] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures. Chapman & Hall, August 2011.

[43] Richard E. Jones and Chris Ryder. A study of java object demographics. In *Proceedings of the 7th international symposium on Memory management*, ISMM '08, pages 121–130, New York, NY, USA, 2008. ACM.

[44] Haim Kermany and Erez Petrank. The Compressor: concurrent, incremental, and parallel compaction. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '06, pages 354–363, New York, NY, USA, 2006. ACM.

[45] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM conference on Language design for reliable software*, pages 128–137, New York, NY, USA, 1977. ACM.

[46] Martin Odersky. The Scala Programming Language. `http://www.scala-lang.org/`, 2010.

[47] Bill McCloskey, David F. Bacon, Perry Cheng, and David Grove. Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors. Technical report, 2008.

[48] Phil McGachey, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Vijay Menon, Bratin Saha, and Tatiana Shpeisman. Concurrent gc leveraging transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 217–226, New York, NY, USA, 2008. ACM.

[49] Mark Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, PODC '97, pages 219–228, New York, NY, USA, 1997. ACM.

[50] David A. Moon. Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 235–246, New York, NY, USA, 1984. ACM.

[51] Takeshi Ogasawara. Numa-aware memory manager with dominant-thread-based copying gc. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 377–390, New York, NY, USA, 2009. ACM.

[52] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. pages 129–140.

[53] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '08, pages 33–44, New York, NY, USA, 2008. ACM.

[54] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[55] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. Aug 1995.

[56] Robert A. Shaw. *Empirical Analysis of a Lisp System*. PhD thesis, Stanford University, 1988. Technical Report CSL-TR-88-351.

[57] Fridtjof Siebert. Concurrent, parallel, real-time garbage-collection. pages 11–20.

[58] Fridtjof Siebert. Limits of parallel marking garbage collection. In *ISMM*, pages 21–29, 2008.

[59] Patrick Sobalvarro. A lifetime-based garbage collector for lisp systems on general- purpose computers. Technical report, Cambridge, MA, USA, 1988.

[60] SPEC. Spec: The Standard Performance Evaluation Corporation. `http://www.spec.org/`, 2010.

[61] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple reservations and the oklahoma update. *IEEE Parallel Distrib. Technol.*, 1:58–71, November 1993.

[62] Gil. Tene, Cliff. Click, Michael. Wolf, and Ivan. Posva. Memory Management, 2006. US Patent 7,117,318.

[63] Gil Tene, Balaji Iyengar, and Michael 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.

[64] Mustafa M. Tikir and Jeffery K. Hollingsworth. Numa-aware java heaps for server applications. IPDPS '05, pages 108.2–. IEEE Computer Society.

[65] Martin T. Vechev, Eran Yahav, and David F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '06, pages 341–353, New York, NY, USA, 2006. ACM.

[66] P. R. Wilson and T. G. Moher. A "card-marking" scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *SIGPLAN Not.*, 24:87–92, May 1989.

[67] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 1–116, London, UK, 1995. Springer-Verlag.

[68] Ming Wu and Xiao-Feng Li. Task-pushing: a scalable parallel gc marking algorithm without synchronization operations. In *IPDPS*, pages 1–10, 2007.

[69] Craig Zilles and Ravi Rajwar. Implications of false conflict rate trends for robust software transactional memory. In *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, IISWC '07, pages 15–24, Washington, DC, USA, 2007. IEEE Computer Society.