The Collie: A Wait-Free Compacting Collector

Balaji Iyengar
Azul Systems Inc
balaji@azulsystems.com

Gil Tene
Azul Systems Inc
gil@azulsystems.com

Michael Wolf
Azul Systems Inc
wolf@azulsystems.com

Edward Gehringer
North Carolina State University
efg@ncsu.edu

Abstract

We describe the Collie collector, a fully concurrent compacting collector that uses transactional memory techniques to achieve wait-free compaction. The collector uses compaction as the primary means of reclaiming unused memory, and performs "individual object transplantations" as transactions. We introduce new terms and requirements useful for analyzing concurrent relocating collectors, including definitions of referrer sets, object transplantation and the notion of individually transplantable objects. The Collie collector builds on these terms and on a detailed analysis of an object's legal states during compaction.

Collie uses a combination of read barriers, write barriers and transactional memory operations. Its read-barrier supports fast, direct object referencing while using a bound, constant time, wait-free triggering path. Collie thereby avoids the constant indirection cost of Brooks [9] style barriers or handle-based heaps [25]. Collie is demonstrated using speculative multi-address atomicity [11], a form of hardware transactional memory supported by the Azul Vega architecture [2].

We evaluate the Collie collector on the Azul platform, on which previous concurrent collectors such as the Pauseless Collector [12] and its generational variant [30] have been commercially available for several years. We discuss Collie's performance while running sustained workloads, and compare it to the Pauseless collector on the same platform. The Collie collector provides significant MMU [5] improvements even in the 1-msec time windows compared to the Pauseless collector. At the same time, it matches Pauseless in throughput and in the ability to scale to large heap sizes.

We believe that the Collie collector is the first garbage collector to leverage hardware-assisted transactional memory. While Collie directly leverages Vega's speculative multi-address atomicity feature (SMA) [11], its design can be easily adapted to other hardware-assisted transactional memory systems. Specifically, the upcoming Intel TSX instruction set extensions [21] include capabilities similar to SMA. We expect Collie to be easily implementable on future commodity servers based on Intel Haswell processors and following processor generations.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Dynamic storage management; Concurrent programming structures; D.3.4 [*Processors*]: Memory management (garbage collection); Run-time environments; D.4.2 [*Storage Management*]: Garbage collection; Virtual memory

General Terms Algorithms, Design, Performance.

Keywords Transactional memory, Atomicity primitives, Concurrent garbage collection, Read barrier, Virtual memory, Runtime systems, Memory management, Compaction, Minimum mutator utilization.

1. Introduction

Managed runtime environments, such as Java, have been grappling with an abundance of cheap compute and memory capacity for some time now. With enterprise application responsiveness expectations that are often in the 10s of milliseconds or lower, scaling runtime instance memory to levels readily available in server platforms has been a challenge. In systems where guarantees or bounds on response time behavior is expected, such as soft or hard real time environments, the challenge is even more pronounced. Along with limitations in practical instance memory size, other scale metrics such as object allocation and access throughput rates are limited when response time requirements are imposed.

The use of stop-the-world garbage collection presents a fundamental fault line in the design of managed runtimes. Garbage collection algorithms in which application threads are stopped for significant periods of time while memory is being reclaimed are clearly in direct conflict with the dual goals of maintaining application responsiveness and scaling application memory instance sizes. As heap sizes grow to 10s and 100s of gigabytes, such stop-the-world techniques become untenable for online processing application. Historically, server growth trends have motivated the design and implementation of several concurrent garbage collection algorithms [3, 12, 23, 26, 30] that all strive to recycle memory without stopping the application, as well as hybrid implementations where at least some of the collection work is concurrent [14].

Controlling fragmentation is an essential part of any garbage collection algorithm meant for sustainable, long running environments. Early concurrent collector implementations [14] delayed compaction, and provided fragmentation control through fallbacks to monolithic stop-the-world compaction modes. Incremental stop-the-world algorithms that attempt to break the monolithic nature of full heap compaction by identifying and compacting parts of the heap in a sequence of incremental pauses [15] have also evolved. However, as heap sizes, live set sizes and allocation throughput all continue to grow, and required response time bounds continue to shrink, implications of such stop-the-world pauses will become unacceptable for most application domains.

Techniques for maintaining responsiveness by allowing concurrent application execution while combating fragmentation span a wide range. Most fall into one of two key categories: completely eliminating the need for compaction, or using concurrent compaction. Eliminating the need for compaction is most often achieved by capping the size of individual memory-contiguous objects, and breaking up objects that require larger sizes into non-contiguous sub-parts (arraylets, objectlets, spines, etc.) [4, 28]. Such no-compaction techniques are often used in real time environments, and are effective at eliminating external fragmentation, but often involve a significant mutator performance overhead due to common access into potentially non-contiguous structures. Concurrent compaction techniques relocate objects in the heap while the application executes concurrently. This introduces synchronization issues between the application threads and the garbage collector, such as the need to ensure correct mutator access through object references that may refer to objects that have been, are being, or are about to be relocated in memory. Concurrent compaction techniques use varying means to grapple with these synchronization issues.

Concurrent compacting collectors [4, 12, 20, 23, 30] often introduce artifacts that impact mutator utilization [5] and performance. These include potentially high barrier [20] or other fast-path costs, such as in the case of Brooks style barriers [4] and handelized-heaps [25]. Other performance limiters include frequent barrier triggering during collector phase shifts [12, 30], and a variable [12, 30] and sometimes unbound [23, 26] amount of time spent in the triggered barrier handling code, for example when the mutator may be required to perform cooperative relocation of object contents in order to handle a triggered barrier. These mutator utilization impacts can have an adverse effect on application throughput and increase jitters in application response times. Supporting ever higher application throughput and ever improving consistent response times while concurrently compacting the heap continues to be the holy grail of garbage collection. We expect that concurrent compactors will see challenges in maintaining expected mutator utilization levels during compaction if they remain prone to high barrier triggering rates and large or potentially unbounded work required for triggered barrier handling.

We believe that advancements in synchronization techniques, such as the use transactional memory, can be used to alleviate some of the limitations that we expect concurrent compactors to run into. In the past couple of decades, transactional memory [18] has emerged as an alternative, potentially scalable synchronization technique on modern hardware systems. Transactional memory typically provides multi-address atomicity semantics, spanning the spectrum from pure hardware implementations [11] to pure software implementations [27], with various hybrid variations [13]. Commercially available hardware assisted transactional memory systems have been available from Azul Systems since 2005 [11]. More recently IBM has been shipping Power cores with hardware transactional memory capabilities in the BlueGene/Q system [16]. While such hardware capabilities have not been available in mainstream servers until now, this has recently changed with Intel's recent anouncement that the upcoming Haswell processors will include transactional memory [21]. It is now expected that hardware transactional memory will become a mainstream feature in commodity servers in the 2013-2014 timeframe.

Our goal in this paper is to take a fresh look at concurrent compaction issues while keeping in sight the ever-increasing responsiveness and scale objectives mentioned above. We describe the Collie collector, a new approach to concurrent compaction that focuses on individual object relocation and leverages transactional memory techniques to provide a bounded cost to triggered barrier operations. In building to the Collie design, we take a general look at concurrent compaction in section 2, and discuss the object transplantation abstraction and its associated correctness requirements. We introduce the concept of individual object transplantation in section 3, and lay out the requirements for maintaining the individual object transplantation quality. In section 4 we build a theoretical, though not quite practical, barrier free algorithm based on this new concept. In section 5, we evolve to the full blown Collie collector– a practical, wait-free implementation of a concurrent compacting garbage collector based on the individual object transplantation concept that leverages read/write barriers and transactional memory.

The contributions of this paper are:

1. We introduce the concept of an "individual object transplantation", the state associated with it, and a set of constraints based on the linearizability [19] framework that are necessary for correct concurrent transplantation.
2. We introduce the first (theoretical, though not practical) barrier-free compacting algorithm.
3. We introduce the first wait-free read barrier with a bounded, constant-time triggering path. The read barrier is self-healing [12] and avoids the constant indirection cost of Brooks [9] style barriers in the fast path.
4. We present an evaluation of our implementation on the Azul Vega platform, and compare Collie with the Pauseless Collector [12] running on the same platform.

We implemented Collie within a fully functional Azul JVM running on an Azul Vega-3 model 7340 appliance containing 432 cores and 384GB in a symmetric memory configuration [2]. Previous collectors such as Pauseless Collector [12] and its generational variant [30] have been shipping commercially on the same platform for several years now. While Collie directly leverages Vega's speculative multi-address atomicity feature [11], its design can be easily adapted to other hardware-assisted transactional memory systems. Specifically, the upcoming Intel TSX instruction set extensions [21] include capabilities similar to the Azul Vega SMA feature. We expect Collie to be easily implementable on future commodity servers based on Intel Haswell processors and following processor generations.

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 paper 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 [30]. 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 paper 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 [6, 12, 23, 30] 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 [20] 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 to obtain the object's to-space pointer. In wait-free copying variations, concurrent accessors may race and concurrently attempt to copy the same object's contents without waiting, where a "winning" copy is decided via an atomic operation. Collectors that use the Brooks style indirection barrier [7, 9] always traverse a forwarding pointer that 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 [6] and LVB-style [12, 30] barriers. Varying implementations of direct-access barrier implementation exist including, hardware read barrier instructions [12], inlined code that checks for invariants [30] and virtual memory protection techniques [12, 23].

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* [19] 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 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 subhistory 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.

[1] This includes mostly concurrent compacting collectors such as the Baker collector [6] as well as fully concurrent compacting collectors such as the C4 collector [30].

[2] According to the Java language spec [20], it is not necessary that the writes to different versions of the objects 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 [22].

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 [20], or relocate + remap collectors [23, 30]). 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 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 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 [6] style algorithms, for example, meet all the linearizability requirements by assuring that threads always obtain the to-space pointer to the object and 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 resulting during the copy phase ensures that no object transplantation set will include any from-space references. Another example is the Sapphire collector [20], 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, and will ensure that no references pointing to from-space locations exist at the end of a compaction phase thereby meeting the second requirement.

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 5 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 completion of a GC cycle. For the purposes of the new algorithms we describe in this paper, 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:

1. 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 two additional limitations on the legal sequential specification:

1. At the beginning of transplantation, no thread stack/register references exist in the object's referrer set.
2. 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 5.

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 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

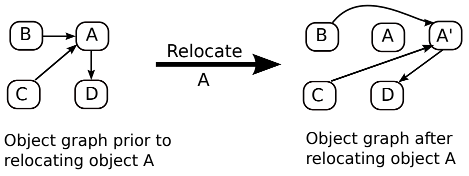
B
A
C
D
Relocate
A
B
A
A'
C
D
Object graph prior to relocating object A
Object graph after relocating object A

Figure 1. Object Graph Illustrating Object Relocation

hand side of figure 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 sequences that violate the legal sequential specification for an individually transportable object: Consider a partial sequence of operations: $[..., Transplant(A), Copy(A, A'), stackRefA1 =$

$ReadRef(A), OK(Copy(A, A')), OK(ReadRef(stackRef - A1)), ...]$. This sequence violates the legal sequential specification since ReadRef(A) creates an additional reference, $stackRefA1$, 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 $stackRefA1$) can result in inconsistent from- and to-copies of the object contents, thereby violating the first condition listed above: $[..., stackRefA1 = ReadRef(A), OK(ReadRef(stackRef - A1)), Transplant(A), Copy(A, A'), Write(stackRefA1), OK(Copy(A, A')), OK(Write(A)), ...]$. Consider this sequence: $[..., stackRefA1 = ReadRef(A), OK(ReadRef(stackRef - A1)), Transplant(A), Copy(A, A'), OK(Copy(A, A')), Up - dateRefs(A, A'), OK(Transplant(A)), Read(stackRefA1), OK(Read(stackRefA1)), ...]$; this is an illegal sequence because the lifetime of stack-reference, $stackRefA1$, spans the transplantation operation, and a reference to the object's from-space location exists after transplantation. On the other hand, the following elaborate sequence involving reads and writes of both object contents and references meets all the limitations for object A to be individually transplantable: $[Trace(), stackRefA1 = ReadRef(A), OK(stackRefA1 = ReadRef(A)), Read(st - ackRefA1), Write(stackRefA1), OK(Read(stackRefA1)), OK(Write(stackRefA1)), OK(Trace()), ReadRef(stack - RefA2), OK(ReadRef(stackRefA2)), Write(stackRef - A2), OK(Write(stackRefA2), Transplant(A), Copy(A, A'), OK(Copy(A, A')), UpdateRefs(A, A'), OK(Transplant(A)), stackRefA1 = ReadRef(A'), OK(ReadRef(stackRefA1), Read(stackRefA1), Write(stackRefA1), WriteRef(stack - RefA1), ...]$

4. Barrier-Free Compaction

We focus our discussion here primarily on concurrent compaction, since concurrent marking is widely discussed [12, 14, 15, 30], and we believe that a highly performant, wait-free, and precise wavefront concurrent marker is essentially a solved problem with the C4's single-pass marker [30]. 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() and WriteRef() 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() and WriteRef() operations all use a lock on the object's from-space copy to synchronize their operations, but such locking would constitute barriers on these mutator operations. We base our algorithm on the use of an Azul style implicit transactional-memory implementation [11], 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 vectors to a transaction failure-handler routine. Concurrent reads of a transaction's "read" data set do not terminate the transaction. 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 point 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, completing the transplantation.

However, there is a 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, concurrent reads or writes of the object's contents 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. Figure 2 reflects the pseudo-code for relocating a single object in the barrier-free algorithm.

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

Figure 2. Barrier-Free Compaction

There are some key takeaways from 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.

5. The Collie Collector Algorithm

We introduce the Collie Collector: a wait-free 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 [30] 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 [30], a write barrier, a checkpoint operation [12], a copy operation, and a transactional transplantation operation.

5.1 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 such that they point to the mirrored-to-space. This can be done in a straight forward 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.

5.2 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 C4 [30] concurrent mark phase, not discussed here), 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 3 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 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.

```
triggered_read_barrier_handler(address, oldValue) {
  mirror = mirror(oldValue)
  AtomicCompareAndSwap(address, oldValue, mirror);
  return  *address
}
```

Figure 3. Pseudo-code for read-barrier trap code

6. 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 phase which already involves walking through the entire heap graph.

[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.

6.1 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 identical to that of the C4 collector [30]; we don't reiterate it here, focusing instead on the changes necessary for building the per-object heap-stable referrer sets.

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 relocators such as Staccato collector [7] and the Chicken collector [26], 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 [15]. We discuss a workaround for the degenerate cases where the popular objects form a significant percentage of the live set.

6.2 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 [30] 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.

6.2.1 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 [12] 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.

At the end of the checkpoint, 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*.

6.2.2 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 maintains the integrity of referrer sets that include references from this object. These referrer-set updates only need to be synchronized against collector activity, and not against the mutator.

[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" such that no compaction would 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

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 [30], 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"*.

7. Discussion and Implementation

The Collie collector is a concurrent compacting collector based on the notion of an individual object's transplantation state. While an object's transplantation state includes the object's contents as well as all references to it, the collector requires only the atomic update of each individual object referrer set when relocating the object to a new location, and allows object contents to be copied non-atomically. Collie relies on transactional memory techniques to implement the atomic state transitions across multiple heap addresses. The transactional memory use is very constrained since only an individual object's referrer set gets updated inside each transaction, and the common individual object referrer set has few elements in it. This results in a high percentage of successful transactions and makes the algorithm amenable even for restricted capacity implementations of multi-address atomicity such as multi-word-CAS [1, 17].

The use of transactional memory on the collector side simplifies the mutator's triggered read-barrier handling code to a single atomic CAS followed by an unconditional read, thereby bounding the read barrier cost. The algorithm does not require any stop-the-world pauses; only checkpoints are needed to ensure coordination with the mutator.

The aborting read-barrier can potentially 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, and 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 [7] and their suggested solution of increasing page density of these pages will work for us as well.

As described, the Collie algorithm is a full-heap collector. However, it can easily be extended to be generational along the lines of C4 [30], which shares its key barrier and synchronization techniques. While generational collection is an almost absolute requirement for a production quality garbage collector on modern hardware, incorporating one into the Collie is orthogonal to our work.

The algorithm is i,mplemented in the production quality Azul JVM [12]. While the algorithm is truly concurrent and does not require any global safepoints, our current implementation does rely on safepoints due to practical runtime concerns, e.g. system-dictionary scanning, class unloading, monitor deflation etc.

8. 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 elements of the collector that 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 barrier triggers. We intend to evaluate and confirm that the collector is able to provide high responsiveness while at the same time being able to sustain high throughput.

The collector has some unique facets such as the individual object referrer set and the 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 [7] and Chicken [26]. We evaluate the effect of these design choices to ensure that they do not adversely affect the collector's ability to perform its primary function, which is to recycle memory. We use a set of standard benchmarks for our experiments: the SPECjvm98 [29] suite, the Dacapo [8] suite, and the SPECjbb2000 benchmark [29]. We run all our tests on an Azul Systems Vega-3 model 7340 appliance– an 8-socket (432-core) appliance with 384GB of physical memory [2]. We compare all our results to a modified version of Pauseless collector [12], a full heap collector running in a single-threaded mode. This allows us to make an apples-to-apples comparison, since both collectors were implemented inside the same JVM, are single-threaded, are full heap collectors, use the same LVB style read barrier (although the triggered barrier 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. We evaluate the behavior of Collie collector's unique properties in section 8.1, and follow that with an evaluation of Collie's latency 8.2 and throughput 8.3 behaviors.

8.1 Evaluating The Collector Properties

The Collie collector builds a per-object referrer set which is then atomically updated during object transplantation. Objects that are referred to by more references than can be tracked by the referrer set allotted for the object are deemed "popular". Pages containing popular objects are deemed popular-pages and are not compacted. The space allotted for object referrer sets directly affects the percentage of popular objects, thereby affecting the percentage of objects transplanted by the Collie collector. The space allotted for object referrer sets also has an effect on the percentage of aborted transplantations, as more popular objects are more likely to experience transplantation aborting accesses. Larger per-object referrer sets allow the collector to compact more of the heap, thereby exposing it to more aborts as well.

The aggregate effective space overhead needed for referrer set storage will depend on the specific mechanisms chosen to implement per-object referrer sets. Efficient mechanisms are possible, in which storage requirements would be approximated by the number of actual non-null references in the heap that do not point to popular objects. Similarly, the global space allotted can be easily capped. We do not explore specific referrer set storage mechanisms in this paper. Instead, we study the effects of capping individual object referrer set sizes on the % of garbage collected, popular pages, and pages aborted; independent of how the referrer set itself is implemented. Our implementation allows us to place an arbitrary cap on individual object referrer set size. We vary the size of this cap in our experiments and measure the effects on the collector properties we present.

Figure 4 plots the amount of garbage reclaimed by the collector at varying referrer set cap sizes. The amount of garbage reclaimed is represented as a percentage of the garbage found. As expected, the result show that 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. For most of the benchmarks run, the largest improvement in the amount of garbage reclaimed occurs when varying the cap size from 1 to 2 tracked references. A referrer set size of 2 tracked references allows Collie to reclaim over 80% of the garbage found in most cases.

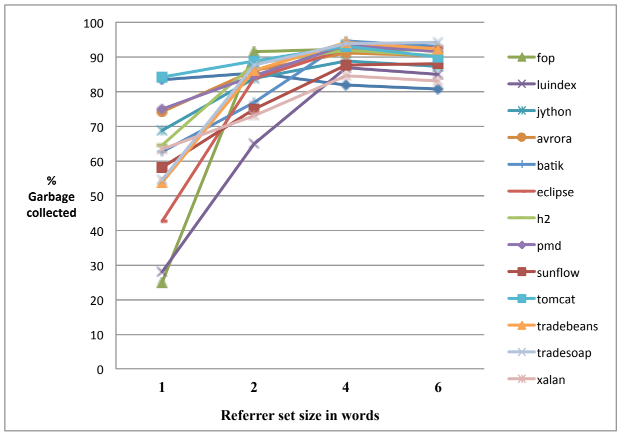

Figure 4. % of Garbage collected versus referrer set size

Figure 5 plots the percentage of popular pages at varying referrer set cap sizes. The percentage of popular pages is computed from the ratio between the number of popular pages and the size of the set of all pages that contain live object. This ratio is not dependent on heap size. As expected, the percentage of popular pages drops as the referrer set cap size increases. The largest reduction in popular page percentage occurs when varying the cap size from 1 to 2 tracked references.

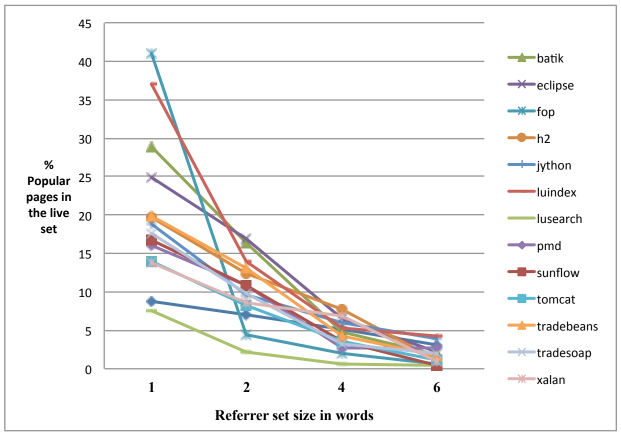

Figure 5. % of Popular pages versus referrer set size

It is interesting to correlate figures 4 and 5. For the smaller referrer set cap sizes of 1 or 2 tracked references, Collie shows a high percentage of popular pages but at the same time is able to reclaim a high percentage of the garbage found. This implies that popular pages are relatively dense in the tests performed, and the live set mostly resides in these pages. Thus the collector would not benefit as much from relocating these pages.

The Collie collector's use of an aborting read barrier has the side effect of creating GC induced fragmentation in non compacted pages. It is important to measure the percentage of pages that had have their compaction aborted. Figure 6 plots the percentage of pages that had their compaction aborted due to mutator interference at varying cap sizes. The percentage of pages aborted is computed from the ratio between the number of aborted pages and the number of pages selected for compaction by the collector. The results indicates a steady, slow growth in the percentage of aborted page-compactions as the per-object referrer size cap increases. This is expected since larger referrer sets imply a higher chance for collision with mutator activity. The percentage of pages aborted still remains a small percentage even at referrer set size cap of 6, allowing the collector to successfully reclaim most of the garbage found.

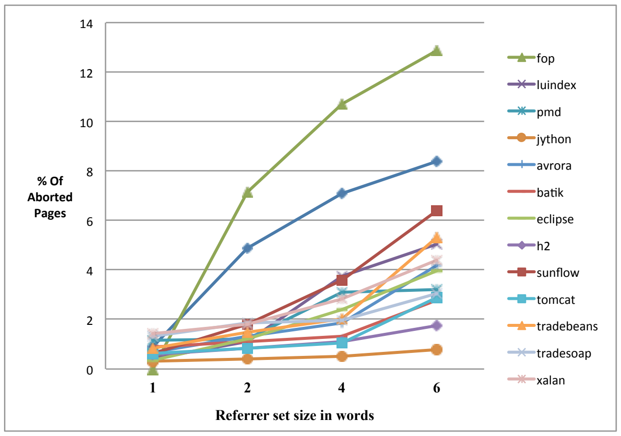

Figure 6. % of Aborted pages versus referrer set size

All runs were performed with a 512MB heap and the GC heuristic set to run continuous, back-to-back GC cycles. The back-to-back collections make our results relatively insensitive to heap size. We experienced variance of less than +/- 3% across a wide range of heap sizes and runs.

8.2 Latency Evaluation

Collie's main goal is to provide high mutator responsiveness during compaction. We measure the minimum mutator utilization (MMU) metric [5] to demonstrate application responsiveness. We compare Collie to the Pauseless [12] collector. Since the implementations of both collectors use a similar marking algorithm and identical phase transitions, we focus our measurements on the MMU achieved specifically during the concurrent compaction phase, as MMU effects during other collection phases are identical for the two collectors. MMU for the Pauseless collector has been studied elsewhere [12, 30].

Table 1 indicates the MMU percentages for compaction phases of the Collie and Pauseless collectors over various time windows ranging from 1ms to 200ms. Each column contain two values, the first reflects the MMU for the Collie collector for the given time window while the second number reflects 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 expected, the Collie collector shows improved MMU across the entire range of time window sizes, with improvements becoming more significant as the MMU time window size

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
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	83/52	86/77	91/87	96/94	99/97
xalan	80/22	86/39	89/64	94/72	97/82	99/87
pmd	78/32	81/45	83/59	87/69	92/72	93/88
eclipse	70/16	82/43	84/49	87/58	89/75	92/82
h2	76/23	83/47	81/52	88/69	90/78	93/83
tradebeans	79/29	87/43	86/74	91/83	96/94	99/97
tradesoap	81/23	86/48	86/69	93/79	97/92	98/96

Table 1. MMU % at different time windows for the Collie/Pauseless collector

decreases. We believe that these improvements are primarily attributable to the aborting read barrier and the Collie collector's tendency to not transplant the "working-set" of the application.

8.3 Throughput Evaluation

The focus of this paper is on compaction, and specifically on wait-free compaction behavior. As such the latency behavior and MMU metrics of the new algorithm's compaction phase, described in section 8.2, are the main focus of our evaluation. The purpose of the throughput comparison in this section is to provide a sanity check for application throughout during compaction. While our measurements show that Collie improved the overall throughput during compaction when compared to Pauseless [12], it is important to note that our focus with Collie was not on improving the throughput of either the collector or the mutator through wait-free compaction. Such improvements are simply an interesting but expected side effect of the lower read barrier triggering rates and the reduced mutator work on triggered barriers.

We use SPECJbb2000 [29], DaCapo [8], and SPECjvm98 [29] benchmarks to specifically compare application throughput during the compaction phases of the Collie and the Pauseless [12] collectors. In order to do this, we modified the heuristics for both the collectors so that we trigger continuous, back-to-back GC cycles, and changed both collectors to use stop-the-world mark phases. With these changes, we ensure that productive benchmark work is only performed during concurrent compaction, and is unaffected by marking, thus allowing us to make a normalized comparison of the application throughput during compaction.

The back-to-back collections and stop-the-world marker, measurement techniques make both throughput and MMU results relatively insensitive to heap size. We experienced variance of less than +/- 3% across a wide range of heap sizes and runs. The results presented for SPECjbb2000 runs used 32 warehouses and a 30G heap. The results presented for the DaCapo and SPECjvm98 benchmarks were obtained with a 512MB heap.

Table 2 shows the relative application throughput during compaction phase of the Collie collector when compared to the Pauseless collector. As we would expect with lower read barrier triggering rates and cheaper triggered-barrier work, the Collie collector improves on this metric. For reference, Table 3 shows the MMU for different time windows for the two collectors during concurrent compaction. As with the results detailed in section 8.2, the Collie collector shows consistently better MMU than the Pauseless collector for these runs.

Collection Algorithm	SPECjbb2000	SPECjvm98	DaCapo
Pauseless collector	1.0x	1.0x	1.0x
Collie collector	1.15x	1.12x	1.09x

Table 2. Aggregate throughput comparison

Benchmark	at 1ms	at 5ms	at 20ms	at 50ms	at 100ms	at 200ms
SPECJbb2000	81/22	86/45	89/64	91/80	95/81	96/83
SPECjvm98	88/33	93/59	96/76	98/81	99/86	98/91
DaCapo	78/24	85/46	88/68	92/80	94/87	97/91

Table 3. Aggregate MMU % at different time windows for the Collie/Pauseless collector

9. Related Work

Cheng and Blelloch [10], 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(MMU). The Staccato collector [7] is the closest to our work in that it uses an aborting read-barrier to obtain a lock-free mutator behavior. However, Staccato is strongly tied to a Brooks' style barrier, an indirection barrier that suffers from the cost of an extra indirection in the mutator fast-path and requires an extra word in the object header. The Collie algorithm is technically agnostic to the flavor of the read barrier and could be used with a Brooks style barrier; however, the benefits of direct object access in the mutator fast-path with a LVB-style healing barrier are hard to overlook. Compressor [23] is a concurrent compacting collector that uses page protection techniques to achieve concurrent compaction but can suffer from significant trap storms and operating system overheads. Variations of the Azul collectors [12, 30], have been shipping in commercial systems for several years, and have focused on addressing the issue of scaling to modern memory and compute capacities while maintaining consistent response times through the use of concurrent compaction. Collie's improvement in MMU behavior compared to Pauseless shows promise in bringing concurrent collection into lower latency and even more latency-sensitive domains. The use of transactional memory in GC algorithms is relatively new; [24] 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. We believe that Collie is the first garbage collector to leverage hardware assisted transactional memory.

10. Conclusion & Future Work

The Collie collector is a concurrent compacting collector that leverages transactional memory and uses a wait-free aborting read barrier. This work breaks new ground in several ways itemized in section 1, but most importantly it represents the first collector to directly leverage hardware-assisted transactional memory, and to achieve wait-free mutator behavior during concurrent compaction by doing so. With hardware assisted transactional memory poised to become a commodity feature and widely available in servers within a year or two, Collie demonstrates that core issues in managed runtime scalability and responsiveness can be addressed using this new capability. We believe that Collie only scratches the surface of the potential for using transactional memory for systemic managed runtime benefits. Future work includes applying Collie in a generational collection environment and applying Collie to commodity server environments as they become available. The newly introduced concepts of individual object referrer sets and individually transplantable objects can be studied in further detail, and additional uses may be found in other garbage collection algorithms and even in areas outside of garbage collection.

References

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

[2] Azul Systems Inc. Vega 3 Processor. http://www.azulsystems.com/products/vega/processor, 2005.

[3] D. F. Bacon, P. 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. ISBN 1-58113-628-5. URL http://doi.acm.org/10.1145/604131.604155.

[4] D. F. Bacon, P. 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. URL http://doi.acm.org/10.1145/604131.604155.

[5] D. F. Bacon, P. 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. URL http://doi.acm.org/10.1145/604131.604155.

[6] H. G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21:280–294, April 1978. ISSN 0001-0782. URL http://doi.acm.org/10.1145/359460.359470.

[7] P. C. Bill McCloskey, David F. Bacon and D. Grove. Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors. In *IBM Research Report RC24505*, pages 285–298. IBM Research, 2008.

[8] 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, Oct. 2006. ACM Press.

[9] R. 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. ISBN 0-89791-142-3. URL http://doi.acm.org/10.1145/800055.802042.

[10] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 125–136, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: 10.1145/378795.378823. URL http://doi.acm.org/10.1145/378795.378823.

[11] J. Choquette, G. Tene, and K. Normoyle. Speculative multiaddress atomicity, 2006. US Patent 7,376,800.

[12] C. Click, G. Tene, and M. Wolf. The Pauseless GC algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 46–56, New York, NY, USA, 2005. ACM. ISBN 1-59593-047-7. URL http://doi.acm.org/10.1145/1064979.1064988.

[13] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. 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.

[14] D. Detlefs and T. Printezis. A Generational Mostly-concurrent Garbage Collector. Technical report, Mountain View, CA, USA, 2000.

[15] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM. ISBN 1-58113-945-4. URL http://doi.acm.org/10.1145/1029873.1029879.

[16] R. H. et al. The ibm blue gene/q compute chip. *Micro, EEE*, 32(2):48 –60, march-april 2012. ISSN 0272-1732. doi: 10.1109/MM.2011.108.

[17] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *In Proceedings of the 16th International Symposium on Distributed Computing*, pages 265–279. Springer-Verlag, 2002.

[18] M. Herlihy and J. E. 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. URL http://doi.acm.org/10.1145/165123.165164.

[19] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990. ISSN 0164-0925. URL http://doi.acm.org/10.1145/78969.78972.

[20] R. L. Hudson and J. E. 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. URL http://doi.acm.org/10.1145/376656.376810.

[21] Intel Inc. Intel Architecture Instruction Set Extensions Programming Reference. http://software.intel.com/file/41604, 2012.

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

[23] H. Kermany and E. 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. ISBN 1-59593-320-4. URL http://doi.acm.org/10.1145/1133981.1134023.

[24] P. McGachey, A.-R. Adl-Tabatabai, R. L. Hudson, V. Menon, B. Saha, and T. 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. URL http://doi.acm.org/10.1145/1345206.1345238.

[25] S. C. North and J. H. Reppy. Concurrent garbage collection on stock hardware. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 113–133, London, UK, UK, 1987. Springer-Verlag. ISBN 0-387-18317-5. URL http://dl.acm.org/citation.cfm?id=36583.36591.

[26] F. Pizlo, E. Petrank, and B. 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. ISBN 978-1-59593-860-2. URL http://doi.acm.org/10.1145/1375581.1375587.

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

[28] F. Siebert. Realtime garbage collection in the jamaicavm 3.0. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, JTRES '07, pages 94–103, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-813-8. doi: 10.1145/1288940.1288954. URL http://doi.acm.org/10.1145/1288940.1288954.

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

[30] G. Tene, B. Iyengar, and M. Wolf. C4: the continuously concurrent compacting collector. In *Proceedings of the international symposium on Memory management*, ISMM '11, pages 79–88, New York, NY, USA, 2011. ACM. URL http://doi.acm.org/10.1145/1993478.1993491.