Chihuahua: A Concurrent, Moving, Garbage Collector using Transactional Memory

Todd A. Anderson
Intel Labs
todd.a.anderson@intel.com

Melissa O'Neill
Harvey Mudd College
oneill@cs.hmc.edu

John Sarracino
University of California San Diego
jsarraci@cs.ucsd.edu

Abstract

Hardware Transactional Memory (HTM) offers a powerful new parallel synchronization mechanism, but one whose performance properties are different from techniques that it competes with, such as locks and atomic instructions. Because of HTM's differing characteristics, when algorithms based on earlier synchronization mechanisms are adapted to use HTM instead, the performance may be disappointing, sometimes even appearing not to significantly outperform software transactional memory. In this paper, however, we show that HTM, and specifically its strong atomicity property, allows approaches to synchronization that would not otherwise be possible, allowing simpler synchronization algorithms than would otherwise be possible that nevertheless have promising performance.

Specifically, we present a new garbage collector named Chihuahua that is designed specifically for HTM, in which garbage collector threads execute transactionally but the mutator does not. In contrast to other work which has adapted existing parallel collectors to make them transactional, Chihuahua is a transactional adaptation of a serial collector (taken from MMTk in the Jikes RVM).

Although Chihuahua is a proof of concept rather than an industrial-strength, production garbage collector, we believe it highlights opportunities in the design space of garbage collectors and other parallel algorithms that are available in HTM but not available in competing techniques.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - Memory Management (garbage collection)

General Terms Algorithms, Performance, Design, Experimentation

Keywords Transactional Memory, Concurrent Garbage Collection, Virtual Machines

1. Introduction

Garbage collection can be viewed as an intrinsically concurrent problem domain—while a program (a.k.a. *the mutator*) runs, the *garbage collector* must examine memory and determine which objects are no longer reachable and can thus be reclaimed. The garbage collector's task is made challenging by the fact that the mutator can make arbitrary changes to memory even as the collector examines it. One approach to these challenges is to minimize concurrency by adopting a turn-taking approach (the most extreme form being "stop the world collection" where the mutator is suspended for the entire collection process), but although this strategy has the advantage of simplicity, arbitrary lengthy pauses in program execution are unacceptable in many domains. For that reason, numerous concurrent approaches have been proposed, from more fine-grained turn-taking approaches to full-blown concurrent collection [13, 14, 23].

Typically concurrent garbage collectors require complex and potentially heavyweight synchronization mechanisms to mediate access to memory and ensure that both mutator and collector operate correctly. As a result, many programming language implementations avoid concurrent collection and adopt simpler stop-the-world schemes, despite their performance disadvantages.

Transactional memory [9] claims to make challenging synchronization problems easier, and hardware implementations provide this facility with low overheads. Until recently, few mainstream implementations of hardware transactional memory existed, but that situation is starting to change as newer Intel CPUs now include Intel *Transactional Synchronization Extensions* (TSX).

Garbage collection appears to be a good proving ground for hardware transactional memory because the coordination problem is nontrivial and both ease of implementation and performance matter. In particular, a transactional approach allows new implementation options that have not previously been considered. It also provides a testbed where we can determine whether the restrictions imposed by a hardware implementation are acceptable or constitute a challenge to overcome.

In this paper, we introduce *Chihuahua*, a new concurrent garbage collector implemented with Jikes' Memory Management Toolkit (MMTk) [3] that uses Intel TSX. Chihuahua is designed as a minimal extension of a non-concurrent GC algorithm. The thesis behind this design was that, unlike most concurrent collectors, Chihuahua would be sufficiently simple that it could be implemented by a small team of undergraduates as a final-year capstone project.

As a result of our work developing Chihuahua, this paper:

- Describes the desirable properties for hardware transactional memory in the GC domain (Section 2).
- Suggests how these properties can be used to design a concurrent collection algorithm (Sections 3 and 4).
- Describes the characteristics of the GC transactions, including transaction failure causes (Sections 4 and 5).
- Suggests how these characteristics can be use to drive transaction retry strategies (Secction 5).

TRANSACT 2015, June 15, 2015, Portland, OR, USA.

2. Transactional Memory

Transactional semantics for system memory was first envisioned by Herlihy and Moss in 1993 [9]. This concept of transactional memory laid out the promise of performance and programmability benefits. In the absence of transactional memory, the largest atomically manipulable memory size on x86 architectures has been a single word. To create larger synchronized structures, a plethora of locking and non-locking algorithms and data structures have been invented, each implemented in terms of those single-word atomic operations. However, determining the best kind and granularity of synchronization to use and implementing these choices correctly by programmers has proven notoriously difficult. Conversely, an ideal transactional memory system would improve programmability by requiring just one easy-to-use synchronization paradigm, namely transactions themselves, and would largely eliminate considerations of granularity. Such transactions could be *arbitrarily sized*, and would have *strong atomicity* (non-transactional writes would abort any transaction that they interfered with), *no spurious aborts* (would abort only when memory conflicts actually occur) and *eventual success* (repeatedly retried transactions would eventually succeed). All that would be left to the programmer would be a conceptually higher-level task of identifying which operations need to be grouped to produce consistent results and then wrapping those in a transaction. Likewise, in an ideal system, transactions would have no overhead and performance would be increased by ensuring that maximum concurrency is maintained in the system.

In the two subsequent decades, a variety of software transactional memory (STM) systems were developed where transactions were provided by the language runtime. Some simpler STM systems augment transactional reads and writes with some additional bookkeeping but forgo this bookkeeping for non-transactional accesses. As a consequence, these STM systems often lack *strong atomicity*, but offer *arbitrarily sized* transactions, *no spurious aborts*, and *eventual success*. These systems are easy to use from the programmer's perspective but are regarded in the literature as inefficient [2, 8, 18].

More complicated STM systems either implement *strong atomicity* or focus on performance, often through the tuning of various STM settings for a particular workload. These systems are generally reported by the literature ([7, 16, 17, 25]) as an improvement over previous STMs. However, tuning these systems for a fixed application (or conversely, tuning an application to the fixed settings of a system) is relatively difficult and there is still often substantial STM overhead. Thus, despite being relatively faithful to Herlihy and Moss' semantics, STM systems have not seen widespread adoption [5].

Until recently, no commodity processors implemented hardware transaction memory (HTM). Intel has introduced Intel® Transactional Synchronization Extensions (Intel® TSX) in the Intel 4th Generation Core™ Processors [1] to provide HTM. TSX provides two ways to execute code transactionally. The first is hardware lock elision (HLE) which is an instruction prefix backward compatible with processors without TSX support. The second is restricted transaction memory (RTM) which is a new instruction set extension and provides the programmer more flexibility in dealing with failed transactions. While RTM provides strong atomicity, transactions in TSX cannot be of arbitrary size due to architectural limitations.

Similarly, RTM guarantees neither *eventual success* nor *no spurious aborts*—a transaction might never succeed, even in the absence of actual contention, as numerous architectural conditions can cause aborts. Such architectural limitations are likely to be common in any processor providing HTM for the foreseeable future. As a consequence, any system that uses Intel RTM must have a fallback, non-transactional mechanism that accomplishes the same result as the transactional path. These transactional and non-transactional paths are typically co-designed so as to properly interoperate with one another. The performance and therefore the complexity of this fallback path is often less critical, as the frequency with which it is run is much lower than if there were no primary transactional path. From a performance perspective, TSX transactions have some startup overhead equivalent to about 3 compare-and-swap (CAS) operations [20] and thus CAS is often used to synchronize single words. For larger critical sections, TSX offers both improved performance and a simpler concurrency model.

As a programming model, RTM is in many ways the dual of STM; it is relatively efficient and offers *strong atomicity*, while lacking *eventual success*, *no spurious aborts*, and *arbitrarily sized transactions*. RTM can sometimes require somewhat more programmer effort than simple STM since every transactional path requires a corresponding fallback path. However, our experience is that creating an algorithm that uses RTM and accommodates its lack of guarantees is easier than the difficulty of making STM efficient.

3. Concurrent Garbage Collection

Garbage collection is an important feature of modern programming languages that removes the burden (and error proneness) of manual memory management from the programmer by automatically detecting and freeing memory which is no longer referenced by the program. Programs can be conceptually divided into the GC component, which is typically provided by the language runtime, and the mutator component. The mutator threads do all the real work of the application while the GC facilitates the mutators by providing memory management services, primarily memory allocation and reclamation. Thus, time spent in the GC is overhead to the mutators and GCs are typically designed to have low overhead and to interfere with the mutators as little as possible. For applications that are response-time sensitive, such as video or telephony, this low interference implies minimizing the amount of time that mutators are paused to cooperate with the GC. Pauses on the order of 10ms (or less) are required for such applications so as not to create noticeable artifacts to the user. This maximum pause time is difficult for nonconcurrent GCs, which typically have the property that all reachable objects must be scanned while the mutators are stopped. Even generational GCs, where there is a small "ephemeral" generation that is collected frequently and quickly, must also occasionally scan for all reachable objects. Current application trends, such as larger working set sizes, larger heaps, larger memory sizes exacerbate these difficulties. Concurrent GCs, in which mutators run at the same time as the GC, can dramatically decrease pause times which results in a smoother and more consistent user experience. In concurrent GCs, mutators may be paused briefly to determine the starting points for garbage collection (a.k.a. *GC roots*) but are then restarted and run concomitantly with the GC threads as those GC threads scan the heap to locate reclaimable memory.

In addition to pause times, mutator throughput remains critical. Mutator throughput can be negatively effected by a GC if the GC allows the heap to become fragmented. A fragmented heap is one is which there are many small areas of free space interspersed with reachable objects. Memory allocation in a fragmented heap is expensive because there is overhead for the GC to switch between allocating in these many different small areas. These small areas also result in more unused space (and therefore more GCs) as it is unlikely that the last allocation in each such space will exactly fill that space. Fragmentation is also bad for subsequent cache locality as objects that are allocated close together in time are likely to be subsequently accessed close together in time. The higher the degree of fragmentation, the more likely it is that related objects will be placed on different cache lines and cause more memory

traffic. A typical solution to prevent fragmentation and to maximize locality is to use a "moving" GC. As its name implies, a moving GC moves objects to create larger free spaces for faster and more cache friendly allocation. Moreover, a moving GC will typically try to maintain object order as it moves objects so as to maximize cache locality.

3.1 The Lost Update Problem

While concurrency and object movement appear to be desirable techniques to minimize mutator interference, all GCs that wish to employ both techniques will encounter and must solve the critical *lost-update* problem, a data-race that we will describe below.

In all concurrent, moving GCs, a mutator may attempt to write to an object that a GC thread is in the process of moving to a new location. Once in its new location, the GC will typically direct mutators to start using the new location rather than the old location, often by installing a forwarding pointer in the old object to point to the new object. Any mutator that sees such a forwarding pointer will follow it and begin to use the new object location. However, there must necessarily be a time delay between when the object is started to be copied and when this forwarding pointer is activated. Consider the example in Figure 1. A GC thread begins to move an object by allocating space for it in the to-space and then copies field A. Then, a mutator writes an update to field A while the GC thread is concurrently copying field B. After copying field B, the GC thread installs the forwarding pointer. Subsequently, any mutator that attempts to read field A and notices that the forwarding pointer is present will redirect its access to the new location of the object. The new location of the object has the value of field A before the mutator's update and so the mutator's update has been lost.

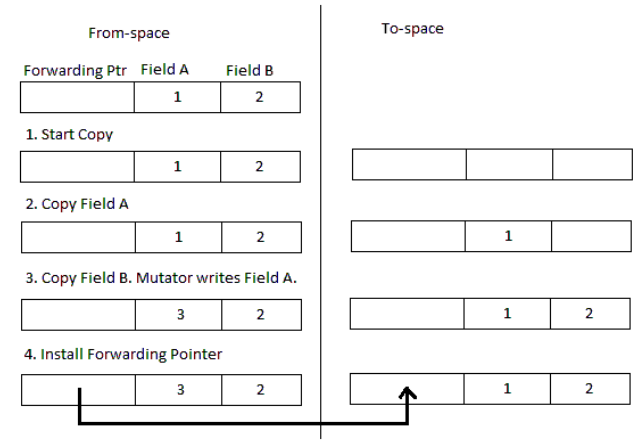

Figure 1. The lost-update problem.

3.2 A Transactional Solution

This *lost-update* problem poses a synchronization issue that existing concurrency models (e.g. CAS signals) do not elegantly solve. The mutator and the collector contend over object bodies, a relatively large portion of shared memory.

In contrast, transactional memory offers a cleaner solution. If both the mutator's writes and the GC's copying operation were atomic, the race condition depicted in Figure 1 would be impossible. This is essentially the path taken by [15], modulo some performance tweaks. However, in practice, the performance overhead of transactions prohibits making mutator writes purely transactional.

4. Chihuahua GC

Our key insight is that transactional mutator writes are only necessary when the TM system is not strongly atomic (e.g. STM). Consider a strongly atomic system (e.g. TSX) in which the GC copying procedure is atomic but the mutator's writes are unconstrained. In such a system, the race condition in Figure 1 would cause the collector's transaction to abort *but leave the mutator unaffected*. In essence, the transactional abort (reported as a synchronization conflict) signals the collector that it missed a mutator's write and that it should retry the copying operation.

This key insight into solving the lost-update problem forms the basis of our concurrent collection algorithm (termed *C*hihuahua) which relies upon the strong atomicity of TSX. Chihuahua is a concurrent, parallel, moving GC implemented in MMTk whose heap structure consists of two semi-spaces plus the normal MMTk immortal and large-object spaces (for objects 4KB and larger). During each GC cycle, all objects in the current semi-space (also called the "from-space") are moved to the other semi-space (called the "to-space"). For clarity, in this section we describe only the aspects of our algorithm that deal with transactional memory or the lost-update problem. Moreover, we limit our discussion here to a non-parallel (i.e., single GC thread) version of our algorithm although our algorithm is extensible to multiple GC threads. Appendix B contains a full description of our GC algorithm, that includes support for multiple GC threads.

4.1 Concurrent Moving Collection

The GC threads and the mutators in Chihuahua collaborate to safely move objects and avoid the lost-update problem. Chihuahua requires that mutators use a write barrier, Section 4.1.3, on reference and non-reference fields during the time in which GC threads are moving objects. Conversely, GC threads perform transactional copying, Section 4.1.1.

Listing 1. Pseudo-code for moving an object with only one GC thread.

```
1  Object * moveObject(Object *object) {
2    for (int tries = 0; tries < MAX_RETRIES; ++tries) {
3      Touch the pages spanned by the object.
4      Execute atomically {
5        Copy object into the to-space, creating a new object.
6        Set object's forwarding pointer to point to the new object.
7      }
8      If the transaction succeeded {
9        Return the new object.
10     } Otherwise {
11       Wait some time before continuing.
12     }
13   }
14   do {
15     Set the object's state to forwarding.
16     Wait for mutators to see the new object state.
17     Copy object into the to-space, creating a new object.
18     Create a pointer to the new object.
19   } while !(Atomically {install the pointer to the new object
20            and update object's state to forwarded.})
21
22   Return the new object.
23 }
```

4.1.1 Transactional Copying

For each object subject to collection, the GC thread attempts to copy the object using the pseudo-code shown in Listing 1. This function is responsible for copying an object (which includes allocating space in the to-space), setting the forwarding word in the old object to point to the new copy, and returning a pointer to the new object.

We first attempt these operations transactionally in lines 4–7.

Should the copy operation fail, we retry transactions up to a pre-specified limit, pausing between transactions. The implementation of this retry mechanism (namely, selecting an optimal retry limit and pause strategy) is described in Section 5.2.

We attempt to avoid a page fault within transactional code (and the subsequent abort) by touching the pages spanned by the object before every transaction in line 3. If a mutator writes to an object here while a GC thread is transactionally copying it, strong atomicity will cause the transaction to abort. The critical lost-update problem is resolved in this circumstance by the subsequent transaction retry which will now copy the updated version of the object.

4.1.2 Fall-back Copying

If our transactional system had the property of *eventual success*, lines 2–9 would be our entire copying algorithm. Unfortunately, TSX does not have this property and so we must provide a non-transactional fall-back path that solves the lost-update problem. Lines 10–16 represent the fall-back portion of our algorithm. This fall-back path is based on the CHICKEN algorithm by Pizlo, et al. [19], in which the collector sets the lower bits of the object's forwarding word to the `FORWARDING` state prior to performing the copy. The GC then ensures that all mutators have a chance to see the new state by performing a handshake in line 12. If the mutator observes an object in the `FORWARDING` state, it changes the object's state in the forwarding word to `WRITTEN`, signaling to the collector that it potentially missed an update. This signal is received via a compare-and-swap failure, which forces lines 15 and 16 to evaluate to false, thus restarting the copying operation. Again, a more complete description of this fall-back path can be found in Appendix B.

An equivalent but more efficient approach to the fall-back path is to batch objects that need to be moved until the size of the batch reaches a certain threshold. Once reached, the GC would set the forwarding state of every object in the batch and then perform a single mutator handshake. In this way, the cost of the handshake would be amortized across a number of objects. After the handshake, the GC thread would copy every object in the batch to the to-space and then individually try to atomically set the objects' states to be forwarded. If such an atomic were to fail (indicating that a mutator had modified the object), that individual object would be included in the next batch and attempted again. This approach has the potential to greatly reduce handshake overhead at the cost of a relatively small increase in the chance that a mutator will write to one of the objects in the batch. Intuitively, the incremental increase in mutator contention as the size of a batch grows is small so these batches could be relatively large in size. Pizlo, et al. [19] suggests possible batch sizes of 1–10 kilobytes but we did not investigate optimum batch size further in this current work.

4.1.3 Write Barriers

Both the transactional and fall-back paths of our algorithm require mutator cooperation to avoid lost updates. To enforce this cooperation, we require all mutator writes to be caught in a level of indirection, known in the GC literature as a barrier. Pseudo-code for our write barrier is shown in Listing 2.

To cooperate with the fall-back path, lines 5–7 detect if the object is in the `FORWARDING` state and if so atomically moves it to the `WRITTEN` state. In lines 8-11, the write barrier determines whether the GC has already moved this object and if so the mutator updates itself to start using the new location of the object. In line 13, the barrier writes the new value of the field into the object. Since the check on line 8 and the write on line 13 are done non-transactionally, there is still a data race with the GC thread's transactional path and there are three possible cases to consider.

- Write before transaction - The write on line 13 completes before the GC thread starts a transaction to move the object. In this case, the GC thread already sees the updated version of the object on its first transaction attempt and the update is not lost.
- Write during transaction - The write on line 13 happens while the GC thread is attempting to transactionally move the object. In this case, strong atomicity will cause the transaction to abort and on retry the mutator's update will be copied and not lost.
- Write after transaction - A GC transaction may have moved the object after line 8 but before line 13. In this case, the mutator's update would be lost (since the write on line 13 would be to the old version of the object) if not for the presence of lines 15–21 in the write barrier. These lines check if the barrier did not previously see the object in the FORWARDED state but now does see the object in the FORWARDED state and by implication that a transaction has moved the object. To prevent the update from being lost, the mutator updates itself to now use the new location of the object (lines 17 and 18) and then re-performs the write, which will now be to the object's new location.

Listing 2. Pseudo-code for write barrier.

```
1  movePhaseWriteBarrier(
2     Object **containing_obj,
3     Object **location,
4     Object *new_value) {
5    If the containing object's state is FORWARDING {
6      Atomically set its state to WRITTEN.
7    }
8    If the containing object's state is FORWARDED {
9      Update the containing object to the forwarded version.
10     Update the location to the field in the forwarded version.
11   }
12
13   Write the new value to location.
14
15   If containing object's state was not FORWARDED before write {
16     If the containing object's state is now FORWARDED {
17       Update the containing object to the forwarded version.
18       Update the location to the field in the forwarded version.
19       Write the new value to location.
20     }
21   }
22 }
```

5. Experiments

To determine the effectiveness of RTM transactions in Chihuahua, we instrumented Chihuahua to record the number of successful transactions and the number and causes of transaction aborts. Following an aborted transaction, a transaction abort code is reported by RTM as a bitmask in the EAX register (see Section 8.3.5 of [11]).

For Chihuahua, we observed only three of these bits set.

- Bit 1 - indicates that the transaction may succeed on retry.
- Bit 2 - indicates that another logical processor conflicted with a memory address that was part of the aborted transaction.
- Bit 3 - indicates an internal buffer overflow which usually indicates a transaction that is too large.

For each aborted transaction, RTM will set zero or more of these bits. During our testing of Chihuahua, only the following bit combinations were observed:

- No bits set - The cause of the abort is not one of the limited set of enumerated abort causes. We report this in the graphs below as *Unknown*. Possible causes of such unattributable aborts include RTM-unfriendly instructions and page faults. To prevent possible page fault aborts, we non-transactionally touch the beginning and end of each object before attempting to move it transactionally. Since our maximum semi-space object size is 4KB, semi-space objects can span only two (4KB) pages and by touching these parts we are guaranteed to touch each such page.
- Bits 1 and 2 - A memory conflict has occurred but the transaction may succeed upon retry. We report this in the graphs below as *Conflict & Retry*.
- Bit 2 - The conflict bit set (but not the retry bit) was observed only a handful of times and is therefore excluded from the graphs.
- Bit 3 - In our initial experiments, we observed some transactions fail with the *overflow* abort code. We reduced the large-object space threshold from 8K to 4K and this eliminated all *overflow* aborts.

All numbers were collected on a 64-bit Ubuntu distribution with an Intel i7-4770 desktop processor. The processor runs at 3.4 GHz, has 4 physical (8 virtual) cores, and has L1, L2, and L3 caches of size 32KB, 256KB, and 8MB respectively.

5.1 Results

While a single GC thread is often sufficient to keep pace with memory allocation, we still wanted to investigate how the parallel version of our algorithm (multiple concurrent GC threads) would perform for larger machines and/or more allocation intensive workloads. Therefore, we collected transaction statistics for numbers of GC threads ranging from 1 to 4 in combination with the benchmarks of the DaCapo suite [4]. We found that these statistics are consistent across all of the suite's benchmarks and so here we only report numbers for the Avrora benchmark. We initially tried to execute each Chihuahua transaction 1000 times or until it was successful, whichever came first. We experimented with this large retry value because, as we note below, there were some transactions that consistently failed with an *Unknown* abort cause and we wanted to know if those transactions would eventually succeed given enough retries. However, here we present figures for only up to 10 of those retries as very few successful transactions were observed after 10 retries. We ran each benchmark multiple times and the transaction numbers reported here are averaged across those runs and across each GC cycle within those runs (so the figures may show fractional transactions).

Results from our initial tests are shown in Figure 2. In these tests, we observed a significant amount of *Conflict & Retry* aborts between GC threads. We also observed a core set of *pathological* transactions (about 0.2% of the original set) that would never succeed regardless of how many times they were retried. Moreover, these transactions generally failed with the *Unknown* abort code, providing no information as to the cause of their failures. We used the Linux *perf* tool to collect TSX-related performance monitoring events and noted a correlation between the number of *Unknown* aborts and the number of "transaction failed due to unfriendly RTM instruction" *perf* events. With this clue, we determined that the cause of these *pathological* transactions was the Jikes semi-space allocator "slow-path." At the beginning of a transaction, Chihuahua first allocates space in the destination semi-space before copying the object there. This allocation usually uses a synchronization-free fast-path but occasionally this allocation uses a slow-path that includes a RTM-unfriendly (i.e., abort inducing) instruction for synchronization.

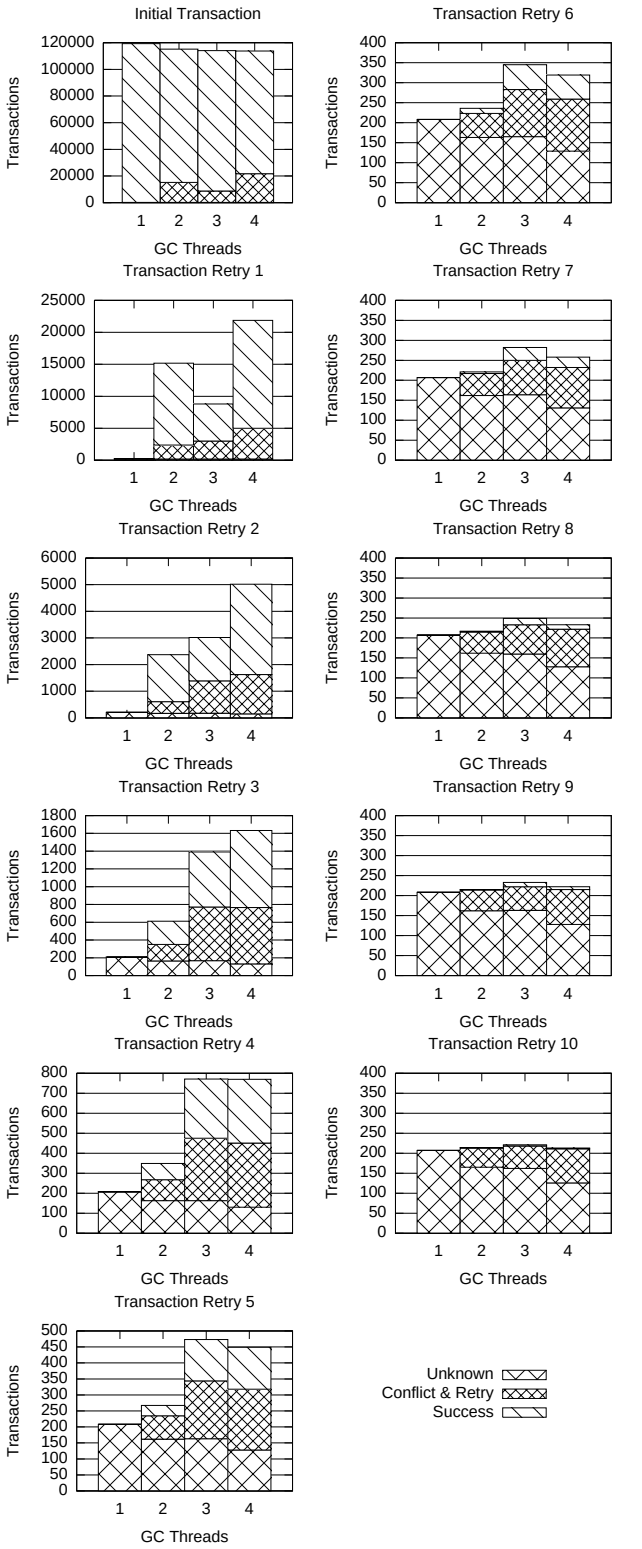

Figure 2. Transaction Retry - Before optimization.

To solve this problem, we modified our algorithm to determine, before starting a transaction, if moving the current object would cause the slow-path allocation code to be invoked. If so, we artificially cause the allocator to move to the next allocation area within the semi-space. Note that this does not increase fragmentation but merely switches allocation areas outside the transactional region rather than within it. Then, when we begin the transaction and call the allocator, we are guaranteed that the transaction will use the allocator fast-path that does not have a RTM-unfriendly instruction. With these changes, all *pathological* transactions were eliminated as were almost all of the *Unknown* aborts and the bulk of the *Conflict & Retry* aborts. Thus, while *eventual success* with TSX is not guaranteed, we did observe *eventual success* in practice.

Number of GC Threads	Initial Transaction Success
1	99.94%
2	99.95%
3	99.93%
4	99.89%

Table 1. Initial Transaction Success Percentage

The percentage of the time that the first transaction attempt now successfully completes is shown in Table 1. We note that about 99.9% of transactions complete in their initial attempt regardless of the number of GC threads so there is little contention between GC threads attempting to move the same object. Figure 3 shows our final set of numbers with our slow-path allocator fix. On average, there is less than one outstanding transaction in the system after three transaction retries. For one GC thread, all transactions complete within three retries while it takes four retries for 3 GC threads. For two and four GC threads, the maximum number of retries is 32 and 43 respectively. The only abort code observed is the *Conflict & Retry* abort code.

5.2 Transaction Retry Strategy

In this section, we discuss how to handle transactions that fail in their first attempt. First, we note that with so few transactions failing in their initial attempt, the cost of retrying the remaining transactions is small. This is especially true when considering the relatively high cost of the fall-back path. Also, when we do repeatedly retry Chihuahua transactions, we observe that they will all eventually complete in at most 43 retries. Thus, in Chihuahua we use the simple retry strategy of retrying until all transactions are complete. However, despite not being observed, *pathological* transactions are still possible and so it is still necessary to have some transaction retry threshold at which point the fall-back path will be triggered. Given the evident rarity of *pathological* transactions, the selection of this threshold is not critical and we recommend twice the longest successful transaction retry number.

We speculated that it might happen that a GC thread may attempt to move an object to which the mutators are in a phase of active modification (e.g., updating every element of an array). Each such modification would then cause a GC transaction to abort. If the transaction retry loop were a tight loop, this might cause repeated transaction aborts. In this case, the GC would essentially have to wait for the mutators to be done with the object before the GC could move it. There could also be some potential systemic overhead related to the GC's repeated transaction aborts. Thus, we experimented with adding various delay mechanisms (such as constant delay, delay proportional to the object size, and exponential backoff) inside the transaction retry loop. One might expect that as this delay was increased, that the maximum transaction retry number would decrease. While in general this was true, there were cases where the opposite was observed. There were some indications that perhaps proportional delay could reduce transactional retries by up to 50% but the small number of transaction retries to start with make this result statistically insignificant.

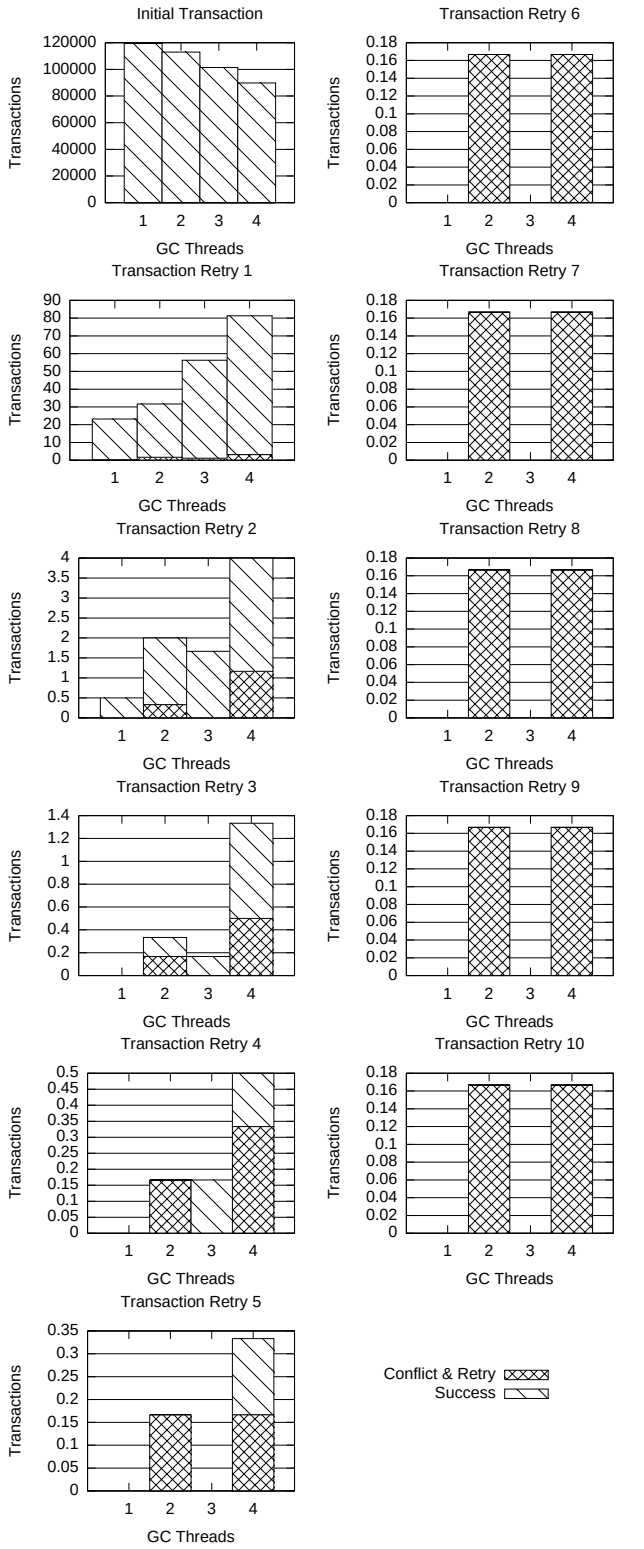

Figure 3. Transaction Retry - With slow-path allocator fix.

6. Conclusions

We have shown how HTM with strong atomicity can be used to solve the lost-update problem in a concurrent, moving GC. We investigated the behavior of Intel TSX in the context of the Chihuahua GC by collecting and analyzing transaction success ratios and failure codes. This investigation led us to make several small changes to the implementation of our algorithm to avoid aborts caused by page faults, RTM-unfriendly instructions, and overflows. After these changes, we saw *eventual success* for each of our collector's transactions. Our initial transaction attempt success rate is largely independent of benchmark and the number of GC collector threads with roughly 99.9% of transactions successfully completing on their first attempt. With *eventual success* and such a high percentage of initial transaction successes, we find that the performance of our collector is not very sensitive to the transaction retry strategy. With the relatively high cost of the fall-back path, we recommend its use only for *pathological* transactions which we speculatively identify as those transactions which have retried more than twice that of the successful transaction with the highest number of retries. These results also confirm that there is not a universally applicable (i.e., context free) transaction retry strategy as our algorithm generally retries until successful whereas other work such as Yoo, et al. [24] used a default of 5 retries and found that up to 80 retries were beneficial for some applications.

7. Future Work

Our work suggests several avenues for further exploration. In particular, there is plenty of opportunity to explore which strategies would increase memory conflicts and cause more interference and which would reduce it. For example, we believe that batching object movement in a manner similar to Ritson [21] would increase the contention rate. Experiments could confirm this outcome and help to determine the best transaction retry strategy in such cases (e.g., debatching the retries). Similarly, although we have provided results for different numbers of GC threads, it would also be useful to explore how to dynamically determine the optimal number of GC threads. We suspect that in many cases, one GC thread (the case that had minimal contention in our experiments) is entirely ample.

8. Related Work

McGachey, et al. [15] describe a block-based (not semi-space), concurrent, moving GC in the context of a software transactional memory (STM) system. Like Chihuahua, they employ a flip phase in addition to the mark and move phases of the GC. In each GC cycle, rather than moving/compacting every object, some selection of blocks are made whose contents are to be moved. The STM system itself required its own set of read and write barriers which were integrated with the barriers required by the GC. Various ways of optimizing this integration of the STM and GC barriers are then described. Unlike Intel TSX, these STM transactions could not fail indefinitely and so no non-transactional fallback path was needed.

McGachey was itself related to the concurrent, moving Sapphire GC [10] in its mark, flip, and sweep functionality. However, Sapphire did not make use of transactional memory and so the copy/-move phase was different and required no read barrier. In Sapphire, no to-space objects were used by mutators until the flip phase. During the copy phase, Sapphire maintains consistency between the from and to-space versions of the same object by mirroring writes to both copies through a write barrier.

Collie [12] is a wait-free, concurrent, moving GC that uses hardware transactional memory provided by Azul Systems Vega processors. These processors also support a read barrier that Collie uses to ensure mutators always access to-space object replicas. During the mark phase, Collie constructs 'referrer sets' of objects (up to some small threshold) which contain references to each object. To minimize transaction size, when Collie moves an individual object, it copies its contents non-transactionally to the to-space. Then, in a transaction, Collie updates references in the object's referrer set to point to the to-space. If the number of referrers is too large or if an object is referenced by a root or is accessed by a mutator while being moved then the object is not moved in this manner but instead the object is virtually copied by mapping its from-space page to the to-space.

Ritson and Barnes [20] collected many TSX performance statistics on an Intel i7-4770 processor and evaluated the suitable of TSX for the implementation of communicating process architectures. One important finding is that TSX transaction overhead is about three times the cost of a compare-and-swap (CAS) so simply replacing CAS with transactions is not beneficial and that transactions can still have reasonable failure rates up to 16KB in size.

Ritson, et al. [21] investigate three possible GC applications of Intel TSX in Jikes/MMTk. The first application was in a parallel semi-space collector to coordinate multiple GC threads to ensure that each object is only moved to the to-space once. Given transaction overhead, they modified this collector to copy to the to-space optimistically so they could transactionally do batch updates to 16 objects' forwarding pointers and state bits at a time but found no benefit from TSX in this application. The second application was for bitmap marking in the MMTk mark-sweep collector. Again, due to transaction overhead, they batched up to 8 bitmap marks locally in a GC thread before committing those marks en-masse to global bitmap. In this case, they found the benefits of the transaction were outweighed by the overhead of the batching process. These results support the notion that merely replacing existing synchronization schemes with transactions does not necessarily lead to improved performance.

The above paper's third application of TSX was for copying objects in an implementation of Sapphire GC in Jikes/MMTk. They determine that 13 objects will fit inside a maximally sized 16KB transaction and explore two ways to form such transactions: inline and planned. Inline starts a transaction while scanning the heap and thus scanning-related reads enter the transaction. Planned copying locally batches up objects to go in the transaction and then starts the transaction and thus there is no extraneous data in the transaction. Planned copying was shown to be superior to inline copying past transaction sizes of about 256 bytes. Finally, concurrent copying with transactional methods was about 60-80% faster than Sapphire's original CAS implementation. Once again, however, this approach is one of replacing an existing CAS-based algorithm with a transactional one. In contrast, our approach relies more heavily on transactional properties like strong atomicity.

Yoo, et al. [24] study how to increase performance of Java monitor locks with TSX and without source or bytecode modifications. In their approach, uncontended (thin) locks continue to be entered with a CAS since CAS has less overhead than a TSX transaction. However, their contended (fat) lock implementation is an integration of a new transactional path (with retries) and the existing JVM fat lock implementation as a non-transactional fallback path. Yoo et al. find that the best retry strategy is this context is application dependent and can be up to 80 retries.. They maintain statistics of transaction success rates for each monitor and use those statistics to switch the monitor implementation on a case-by-case to the original, transactionless implementation if the transaction failure rate of the monitor is too high. Likewise, if lock count is above a threshold, they can switch the monitor to a faster implementation that does not have the overhead of statistics gathering.

Acknowledgments

This work began under the auspices of the Harvey Mudd College *Clinic* program, a capstone project in which a team of four undergraduates tackles a problem of particular interest to an industry sponsor (in this case Intel) in an academic setting. The four students were John Sarracino, Joe Agajanian, Claire Murphy, and Will Newbury. We would like to extend our thanks to everyone at Intel and Harvey Mudd College who helped make the project possible. We would also like to thank James Cownie and Roman Dementiev at Intel for their help in answering various questions we had about the behavior of TSX on Intel CPUs.

References

[1] *Intel architecture instruction set extensions programming reference.*, chapter 8. Intel Corporation, 2012.

[2] A. Baldassin, E. Borin, and G. Araujo. Performance implications of dynamic memory allocators on transactional memory systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 87–96, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3205-7. . URL `http://doi.acm.org/10.1145/2688500.2688504`.

[3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *26th International Conference on Software Engineering*, pages 137–146, Edinburgh, May 2004.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. . URL `http://doi.acm.org/10.1145/1167473.1167488`.

[5] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):40, 2008.

[6] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Sholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. In *Communications of the ACM*, pages 966–975, 1978.

[7] A. Dragojevic. On the performance of software transactional memory. 2012.

[8] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why stm can be more than a research toy. *Communications of the ACM*, 54(4):70–77, 2011.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993. ISSN 0163-5964. . URL `http://doi.acm.org/10.1145/173682.165164`.

[10] 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. ISBN 1-58113-359-6. . URL `http://doi.acm.org/10.1145/376656.376810`.

[11] Intel Corporation. Intel architecture instruction set extensions programming reference, 2012.

[12] B. Iyengar, G. Tene, M. Wolf, and E. Gehringer. The collie: A wait-free compacting collector. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 85–96, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1350-6. . URL `http://doi.acm.org/10.1145/2258996.2259009`.

[13] R. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley & Sons, 1996. ISBN 0471941484 (alk. paper).

[14] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures. Chapman & Hall, 2011. ISBN 978-1420082791.

[15] 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. ISBN 978-1-59593-795-7. . URL `http://doi.acm.org/10.1145/1345206.1345238`.

[16] M. Moir and D. Nussbaum. What kinds of applications can benefit from transactional memory? In *Computer Architecture*, pages 150–160. Springer, 2012.

[17] V. Pankratius and A.-R. Adl-Tabatabai. Software engineering with transactional memory versus locks in practice. *Theory of Computing Systems*, 55(3):555–590, 2014.

[18] V. Pankratius, A.-R. Adl-Tabatabai, and F. Otto. *Does transactional memory keep its promises?: results from an empirical study*. Univ., Fak. für Informatik, 2009.

[19] 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`.

[20] C. G. Ritson and F. R. Barnes. An evaluation of intel's restricted transactional memory for cpas. In *Communicating Process Architectures*, 2013.

[21] C. G. Ritson, T. Ugawa, and R. E. Jones. Exploring garbage collection with haswell hardware transactional memory. In *Proceedings of the 2014 International Symposium on Memory Management*, ISMM '14, pages 105–115, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2921-7. . URL `http://doi.acm.org/10.1145/2602988.2602992`.

[22] G. L. Steele, Jr. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, Sept. 1975. ISSN 0001-0782. . URL `http://doi.acm.org/10.1145/361002.361005`.

[23] P. R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *Proceedings of the International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42, St Malo, France, 16–18 Sept. 1992. Springer Verlag. URL `ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps`.

[24] R. Yoo, S. Viswanathan, V. Deshpande, C. Hughes, and S. Aundhe. Early experience on transactional execution of java programs using intel transactional synchronization extensions. In *Ninth ACM SIGPLAN Workshop on Transactional Computing*, March 2014.

[25] M. Zhang, J. Huang, M. Cao, and M. D. Bond. Larktm: Efficient, strongly atomic software transactional memory. Technical report, Technical Report OSU-CISRC-11/12-TR17, Computer Science & Engineering, Ohio State University, 2012.

Appendices

A. Implementation of Transactions

Hardware transactional memory is key to our algorithm and using it in C code is relatively straightforward. Listing 3 shows an example RTM algorithm in C, written with the inline assembly directive `asm`. The programmer starts a transaction with the `xbegin` instruction (line 1). This `xbegin` instruction will jump to its argument on a transactional failure, so the programmer passes the `tx_retry` label on line 7. Finally, `xend` on line 5 ends the transaction. Lines 4-7 are only reached within a transaction while line 9 is only reached on transaction failure.

Listing 3. C transactional memory example.

```
1   tx_start:
2     asm goto("xbegin %l[tx_retry];": : :"memory"
3              ,"ax":tx_retry);
4     // transactional code
5     asm volatile("xend");
6     // unreached if transaction fails
7     goto done
8   tx_retry:
9     // retry policy and fail-path code
10  done:
```

However, our GC is written in Java because the Jikes Java Research Virtual Machine (RVM) is itself written largely in Java. The Jikes code generator does not yet contain a `__xbegin` compiler intrinsic or the `atomic` keyword used in our algorithm, so we needed a way to work with RTM in Java code. JikesRVM's internals include a way to call short, non-recursive C functions (called *syscalls*) from Java code, which are essentially inlined. Using this mechanism, we exported RTM to high-level code through two function calls, `BeginTransaction` and `EndTransaction`, shown in Listing 4. `EndTransaction` is relatively trivial, as it merely inlines `xend`. `BeginTransaction` starts a transaction and returns a signal for whether the transaction succeeded or failed.

If a transaction fails, this code does not attempt to retry the transaction, so application code is responsible for the retry strategy.

Listing 4. JikesRVM transactional memory interface.

```
1  int BeginTransaction() {
2    tx_start:
3      asm goto("xbegin %l[tx_retry];": : :"memory"
4               ,"ax":tx_retry);
5      return 1; // signal entrance to transaction
6    tx_retry:
7      return 0; // signal fail case
8  }
9
10 void EndTransaction() {
11   asm volatile("xend");
12 }
```

This implementation is straightforward but overall control flow merits some explanation. A naive application might be tempted to logically consider `BeginTransaction` and `EndTransaction` as inlining `xbegin` and `xend`, respectively. This might result in code such as Listing 5, in which `BeginTransaction` and `EndTransaction` delimit a transaction. However, such code is not correct, as control returns to lines 2-3 both when a transaction starts and when a previous transaction fails.

Listing 5. Erroneous JikesRVM transactional memory example.

```
1  BeginTransaction()
2  // transactional code
3  // or is it?
4  EndTransaction()
```

This error can be rectified by using the return value of `BeginTransaction` in a conditional to determine the mode of operation. For example, a conditional pattern such as Listing 6 correctly encodes the semantics (of a single transaction) intended by most applications. Our Chihuahua algorithm moves objects using identical logic.

Listing 6. Correct JikesRVM transactional memory example.

```
1  if BeginTransaction() {
2    // transactional code
3    EndTransaction()
4    // unreached if transaction fails
5  } else {
6    // fail-path code
7  }
```

B. Detailed Chihuahua Algorithm

Chihuahua is a concurrent, parallel, moving GC implemented in MMTk whose heap structure consists of two semi-spaces plus the normal MMTk immortal and large-object spaces (for objects 4KB and larger). This arrangement was chosen for simplicity and expediency as we were able to borrow from the existing non-concurrent MMTk semi-space collector. Our implementation of Chihuahua supports multiple GC and mutator threads. The concurrent GC threads runs continuously and in each GC cycle moves all reachable, semi-space objects from the current semi-space to the other semi-space. While perhaps not practical in a production collector, this semi-space approach of moving every object every GC cycle has the benefit from a testing perspective that it maximizes the amount of object movement and therefore stresses the transactional part of the collector better allowing us to test the correctness of the algorithm and performance of the transactions themselves. Likewise, unless the number of hardware threads is large, it is not usually necessary to have multiple GC threads or to run them continuously (as it can typically free memory faster than it is allocated) but we do so to stress the collector for testing purposes. In terms of object format, the Chihuahua collector requires header space for a forwarding pointer and also uses the two lower bits of this pointer which would otherwise be unused due to alignment. The purpose of these extra bits is described below. The following sub-sections describe the Chihuahua algorithm from the perspectives of the GC thread and of the mutators.

B.1 GC Phases

Each cycle of the concurrent GC thread consists of three major conceptual phases, inspired by McGachey, et al. [15]. Those phases are the mark-scan phase, the move (what McGachey calls the copy) phase, and the flip phase.

B.1.1 Mark Phase

Before starting this phase, we make sure that the mutators have enabled their mark phase write barrier (described in the mutator section). Then, each mutator is individually stopped only long enough to scan its stack and registers to find GC roots. After all roots have been found, the mark phase uses the classic tri-color marking algorithm and invariant [6, 22] to transitively identify all reachable objects in the current semi-space starting from the GC roots. We borrow most of our implementation of this phase from the MMTk concurrent mark-sweep collector's mark phase. The purpose of the write barrier in this phase is to ensure that the tri-color invariant is maintained.

B.1.2 Move Phase

The move phase is responsible for moving objects to the destination semi-space safely and avoiding the lost-update problem. Before starting the move phase, we install the move phase read and

write barriers. Then, for each reachable object identified in the mark phase, the GC thread attempts to move the object to the other semi-space using the pseudo-code shown in Listing 7. In line 2, the GC first reads the old forwarding pointer word which consists of the forwarding pointer and in the lower two bits are flags which indicate whether the object has been forwarded (the 'FORWARDED' bit) or is being forwarded (the 'FORWARDING' bit). If the forwarding word was clear, line 2 also sets the forwarding word to 'FORWARDING'. The forwarded state indicates that the object has been moved to the new semi-space whereas the forwarding state indicates that a GC thread is moving the object.

These lines mediate GC-GC contention in that each object is intended to be moved by exactly one GC thread.

In lines 3-7, we check if the object has or is in the process of being forwarded and if so wait until that is fully completed and return the new location of the object. Lines 3-7 are only strictly necessary for the FORWARDING case when there are multiple GC threads.

Lines 9-20 constitute our transactional copy, in which we retry upon transactional failures.

On line 10, we prevent a possible page fault from aborting the subsequent transaction by reading the pages possibly spanned by an object (see Section 5).

At line 11, the current thread will attempt to move the object transactionally and starts a hardware transaction. If the object has still not been forwarded then in lines 12-14 we allocate room for and copy the object into the destination semi-space and set the forwarding pointer and the FORWARDED bit in the forwarding word of the header of the old location of the object in the current semi-space.

If a mutator writes to an object non-transactionally during this copying process, strong atomicity will cause the GC transaction to abort. If an abort occurs, the transaction retry policy is applied. If the number of transaction retries does not exceed a certain maximum then control reverts to line 10 and the transaction is tried again. During such retries, the GC thread will now copy the updated version of the object and thus prevent the lost-update problem. This policy of the maximum number of transaction retries is examined in Section 5.

The non-transactional fallback path is based on the CHICKEN algorithm by Pizlo, et al. [19]. On line 22, we non-atomically set the FORWARDING bit of the object to be moved and on line 23 update oldFwd to reflect the new state of the forwarding word. Then, on line 24 we wait for all mutators to reach a certain point whereat they are guaranteed to see the FORWARDING bit set if they happen to examine the object's forwarding word.

On line 25, we allocate space for and copy the object to the destination semi-space and on line 26 compute what the new forwarding word of the old object should contain, namely the new address of the object or'ed with the FORWARDED bit. Finally, on line 27, we attempt to atomically set the forwarding word of the old object location to the new one from line 26. This atomic operation will fail if the forwarding word does not have the FORWARDING bit set as contained in oldFwd. As discussed in Section B.2.2, a mutator can atomically clear this FORWARDING bit if it needs to write to the object. In which case, the GC thread repeats the loop so that the copy on line 25 will include the modified data. In this approach, the mutator fast path, in which the FORWARDING bit is not set, is privileged and fast by being free of atomics. Conversely, the GC thread is disadvantaged and is responsible for waiting until mutators are guaranteed to see the FORWARDING bit and for retrying as many times as necessary until moving the object succeeds.

The handshake with the mutators on line 24 is potentially a very expensive operation. One possible optimization, which for lack of time we did not implement, is to batch objects whose transactions fail and to then attempt to move multiple such objects while sharing a single mutator handshake. In such a batch, the forwarding bit of each object would be set, one handshake performed, the copies performed, and a compare and swap on the forwarding pointer of each object performed. All objects for which the compare and swap fails to find the FORWARDING bit set would similarly need to be repeated.

Listing 7. Pseudo-code for moving an object.

```
1  Object * moveObject(Object *object) {
2    oldFwd = readAndAcquireForwardingWord(object)
3    if isForwardingOrForwarded(oldFwd) {
4      while !isForwarded(oldFwd) {
5        oldFwd = readForwardingWord(object)
6      }
7      return getForwardingPtr(object)
8    } else {
9      for (int i = 0; i < RETRY_DEPTH; ++i) {
10       touchMiddleEnd(object)
11       if BeginTransaction() {
12         new_object = allocateAndCopy(object)
13         setForwardingPtr(object, new_object)
14         setForwardedBit(object)
15         EndTransaction()
16         return new_object
17       } else {
18         pauseBeforeRetry(object)
19       }
20     }
21      do {
22       setForwardingBit(object)
23       oldFwd |= FORWARDING_BIT
24       mutatorHandshake()
25       new_object = allocateAndCopy(object)
26       newFwd = new_object | FORWARDED_BIT
27     } while (CASForwardingPtr(object, newFwd) != oldFwd)
28   }
29 }
```

B.1.3 Flip Phase

The flip phase uses the same set of mutator read and write barriers as the move phase. The purpose of the flip phase is to ensure that all references in the program now refer to the destination semi-space. As described in Section B.2.2, some such references will be updated by mutator read/write barriers but most such references will be updated by the flip phase. The flip phase is nearly identical to the mark-scan phase and in our implementation is a modified version of the mark-scan phase in which each thread is temporarily suspended in turn to identify GC roots which are then updated to point to the destination semi-space. Those roots are then used to transitively visit each object in the destination semi-space and to then update all references in those objects to point to the destination semi-space if they do not do so already. After the flip phase is done, the read and write barriers can be eliminated for the mutators until the start of the next GC cycle.

B.2 Mutator Barriers

Different kinds of read and write barriers are required for mutators to cooperate with the GC during the different phases of the GC cycle. For example, during the mark-scan phase of the GC, the mutators require a write barrier on reference fields only. Conversely, during the other phases read and write barriers are required as described below.

B.2.1 Mark Phase Barriers

The only barrier required during the mark/move phase is a write barrier on references. Here we reuse the existing MMTk concurrent

mark-sweep collector write barrier. This barrier ensures that the tri-color invariant is maintained by graying otherwise white objects if they are written into a black object.

B.2.2 Move Phase Barriers

During the move phase, write barriers are required for both reference and scalar fields and a read barrier is required for reference fields. All the write barriers function in the same manner and have the same purpose of preventing the lost-update problem. Pseudo-code for the reference version of the write barrier is shown in Listing 8 (scalar versions are similar). When the write barrier is invoked, it first reads the forwarding word of the object containing the location to be written in line 6. In line 7, the FORWARDING bit is set on the object which indicates the GC is trying to move this object using the non-transactional fallback path. Since the mutator has priority, we atomically clear the FORWARDING bit in line 8 so that the GC thread will fail the CASForwardingPtr in Listing 7 and will try to move the object again. On line 10, we check the object has already been moved. If so, we compute the offset of the location to be written from the base of the object in line 12, modify the reference used to access this object on line 13 and compute the new location to be written by adding the computed offset to the new object location in line 14. On line 17, the barrier then writes the new value into the location. If on line 19 the FORWARDED bit we originally read was not set, then there is the potential that the GC thread has transactionally moved the object since the FORWARDED bit was read in line 7. To test for this case, we read the forwarding word again in line 20 and check the FORWARDED bit in line 21. If the FORWARDED bit is now set, we cannot know whether the mutator's write occurred before or after the GC thread's transaction which moved the object. If the write occurred after the transaction then this would result in a lost-update and so on lines 22-24 we update the containing object and location just as we did in lines 12-14. Then the write of the new value into the location is repeated. Note that if the barrier's first write occurred before the GC transaction and was therefore included in it that this second write will write the same value and be benign. Also note that if a transaction does move the object prior to line 20 that the transaction will have invalidated the cache line on the processor running the barrier and so line 20 is guaranteed to see the FORWARDED bit set.

Listing 8. Pseudo-code for move phase write barrier.

```
1  movePhaseWriteBarrier(
2    Object **containing_obj,
3    Object **location,
4    Object *new_value) {
5
6    oldFwd = readForwardingWord(*containing_obj)
7    if isForwarding(oldFwd) {
8      atomicClearForwardingBit(*containing_obj)
9    }
10   originally_forwarded = isForwarded(oldFwd)
11   if originally_forwarded {
12     offset = location - *containing_obj
13     *containing_obj = getForwardingPtr(*containing_obj)
14     location = *containing_obj + offset
15   }
16
17   *location = new_value;
18
19   if !originally_forwarded {
20     curFwd = readForwardingWord(*containing_obj)
21     if isForwarded(curFwd) {
22       offset = location - *containing_obj
23       *containing_obj = getForwardingPtr(*containing_obj)
24       location = *containing_obj + offset
25       *location = new_value;
26     }
27   }
28 }
```

The move phase read barrier is only applied to reference fields. Its purpose is to ensure that a read following a write to a destination semi-space object will see that write. Pseudo-code for this read barrier is shown in Listing 9. In line 2, the barrier reads the forwarding word for the value currently in the reference field. If that object has been forwarded (line 3) then the reference field is updated to point to the new location of the object in line 4. Finally, the most recent reference field is returned on line 6.

Listing 9. Pseudo-code for move phase read barrier.

```
1  Object * movePhaseReadBarrier(Object **ref_field) {
2    oldFwd = readForwardingWord(*ref_field)
3    if isForwarded(oldFwd) {
4      *ref_field = getForwardingPtr(*ref_field)
5    }
6    return *ref_field
7  }
```

B.2.3 Flip Phase Barriers

The flip phase barriers must ensure that reads and writes occur to destination semi-space objects. While not optimal, this can be simply accomplished by leaving the move phase barriers in place and this is what we do in our implementation. The minimal move phase and flip phase read barriers are identical. However, for the flip phase write barrier, the FORWARDING state is impossible as is the FORWARDED bit changing between lines 6 and 20 and thus lines 7-9 and 19-27 are superfluous in the flip phase.