Towards an Efficient Pauseless Java GC with Selective HTM-Based Access Barriers

Maria Carpen-Amarie
Université de Neuchâtel, Switzerland
maria.carpen-amarie@unine.ch

Yaroslav Hayduk
Université de Neuchâtel, Switzerland
yaroslav.hayduk@unine.ch

Pascal Felber
Université de Neuchâtel, Switzerland
pascal.felber@unine.ch

Christof Fetzer
TU Dresden, Germany
christof.fetzer@tu-dresden.de

Gaël Thomas
Telecom SudParis, France
gael.thomas@telecom-sudparis.eu

Dave Dice
Oracle Labs, USA
dave.dice@oracle.com

ABSTRACT

The garbage collector (GC) is a critical component of any managed runtime environment (MRE), such as the Java virtual machine. While the main goal of the GC is to simplify and automate memory management, it may have a negative impact on the application performance, especially on multi-core systems. This is typically due to *stop-the-world* pauses, i.e., intervals for which the application threads are blocked during the collection. Existing approaches to concurrent GCs allow the application threads to perform at the same time as the GC at the expense of throughput and simplicity. In this paper we build upon an existing pauseless transactional GC algorithm and design an important optimization that would significantly increase its throughput. More precisely, we devise *selective access barriers*, that define multiple paths based on the state of the garbage collector. Preliminary evaluation of the selective barriers shows up to 93% improvement over the initial transactional barriers in the worst case scenario. We estimate the performance of a pauseless GC having selective transactional barriers and find it to be on par with Java's concurrent collector.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**;
• **Software and its engineering → Garbage collection**;

KEYWORDS

Garbage Collection, Hardware Transactional Memory, Java

ACM Reference format:
Maria Carpen-Amarie, Yaroslav Hayduk, Pascal Felber, Christof Fetzer, Gaël Thomas, and Dave Dice. 2017. Towards an Efficient Pauseless Java GC with Selective HTM-Based Access Barriers. In *Proceedings of ManLang 2017, Prague, Czech Republic, September 27–29, 2017,* 7 pages.
https://doi.org/10.1145/3132190.3132208

1 INTRODUCTION

With multi-core systems becoming ubiquitous in the last few years, lightweight synchronization and efficient automatic memory management are more and more demanded in the design of applications and libraries. The GC is the mechanism responsible for automatic memory allocation and recycling in any MRE. Its goal is to identify the chunks of memory that are not used anymore and make them available for further allocation, while disrupting as little as possible the execution of the application. The Java Hotspot virtual machine (VM) is arguably one of the most widely used VMs. It features several types of GCs, all of which need to block all application threads, i.e., *stop the world*, during collection, either for the entire process, or just for certain phases. This sometimes results in significant pauses, that can have a considerable impact on the application performance: the application can spend 14% to 60% of its execution time waiting for the GC to collect, depending on the available memory and other configuration details [5].

We aim to design an efficient pauseless garbage collector in Java. One essential difference from the existing Java GCs is the synchronization of the application threads with the GC threads, using *hardware transactional memory* (HTM). The idea is to replace the complex lock-based synchronization with a lightweight mechanism, that optimistically assumes there are no conflicts and allows the application threads to proceed while the GC is collecting.

As a preliminary step towards our goal, this work presents and evaluates a new type of HTM-based barriers for loads and stores on the application side, namely *selective transactional access barriers*. They represent an optimization over a former design of transactional barriers, that were shown to cause an unacceptable overhead [5]. Therefore, before proceeding with the full implementation of a pauseless transactional GC, we first estimate to what extent selective barriers can reduce the previously reported transactional overhead. Throughout the paper, the overhead is computed with respect to the performance of Java's ConcurrentMarkSweep GC.

We test our optimization on a benchmark suite, as well as in a real-life memory-intensive scenario, involving a client-server database system. Despite using the lowest bound for the duration of a transaction, we still obtain up to 93% less overhead than the former transactional barriers for some applications. Moreover, 5 out of 7 workloads in the client-server configuration show a maximum overhead of less than 5%. These results open the way to implementing a fully-concurrent GC that will guarantee little to no overhead outside GC execution, while during the collections the overhead will be amortized by avoiding GC pauses. A rough estimation of the CPU time of the transactional GC with selective barriers indicates comparable performance to state-of-the-art GCs. Finally, we consider several challenges in implementing the transactional GC and briefly present a potential approach for removing the current lock-based GC synchronization.

2 STATE OF THE ART

This work aims to bring together two essential mechanisms in today's concurrent systems, garbage collection and lock-free synchronization, into a fully concurrent garbage collector based on transactional memory.

2.1 Background

The *garbage collector* is a key component in any managed runtime environment, such as the Java virtual machine (JVM), in particular Hotspot JVM. Its task is to allocate and collect unused memory in a transparent and efficient way. Hotspot's GCs are generational (they split the allocation space into generations and collect them separately) and tracing (they follow all object references, and identify which objects are reachable). Hotspot JVM provides seven different GC algorithms. Out of these, we only focus on their concurrent collector, *ConcurrentMarkSweep* (CMS). CMS is a generational collector that uses a parallel stop-the-world collector for the young generation, called *ParallelNew* (ParNew) GC, and a concurrent low-pause collector for the old generation.

Transactional memory (TM) [11] was first introduced as an efficient and scalable solution to classical synchronization issues. Over the last few years it gained a significant popularity in the research world. In essence, TM is a lock-free synchronization mechanism, that avoids using classical locks by encapsulating the targeted block of instructions in a transaction executed atomically. The software implementation was mostly used in research [7, 10], having a substantial overhead. Hardware TM, on the other hand, was included in commodity hardware since mid-2013 (Intel Haswell processor) together with a new instruction set for atomic manipulation of memory, Intel *transactional synchronization extensions* (TSX). This provides two new synchronization mechanisms: hardware lock elision (HLE) and restricted transactional memory (RTM). In this work we rely on RTM, which provides an entirely new set of instructions that allow for more flexibility in defining transactional regions.

2.2 Related Work

A first attempt at combining garbage collection with transactional memory was made as early as 2008 by McGachey et al. [14]. They devised a concurrent GC algorithm based on STM with soft real-time guarantees. Later, when HTM became part of specialized processors and systems a few proposals addressed the idea of employing it in GC algorithms. The most relevant proposal is the implementation of an HTM-based Java GC on a specialized multi-core CPU built by Azul systems, by Iyengar et al. [12]. This GC algorithm is shown to have better performance than their baseline copying and concurrent collector without HTM. However, it only collects on one core and needs two passes over the reference graph. In mid-2013, Intel made their HTM implementation available to the public in the Intel Haswell CPU, creating a greater opportunity for research in this area. Anderson et al. [2] provide a simple proof of concept GC, built on top of Jikes Research VM. Chihuahua is a minimalistic concurrent, moving GC, that uses HTM for synchronization. The paper focuses more on HTM properties that are used when implementing the GC, failure causes and retry strategies. On the other hand, Ritson et al. [15] propose a performance study on three main GCs in Jikes RVM implemented with HTM. They find that one of the algorithms is considerably improved by HTM, while the other two have similar performance to their original implementation. Our work complements these contributions by designing an optimized concurrent HTM-based collector built on top of a real-life Java GC.

3 OUR APPROACH

Previous work [6] illustrated the impact of garbage collection on application performance in two situations: on a widely used benchmark suite (i.e., DaCapo benchmarks [3]), as well as on a real-life large-scale application, the Apache Cassandra database server [1]. The most relevant results indicate that even the optimized Java concurrent collectors, such as CMS and G1 [9], can disrupt application execution with considerably long stop-the-world pauses when confronted with memory-intensive applications. We propose an optimization to an HTM-based concurrent GC algorithm that eliminates the need for stop-the-world pauses, which allows it to also maintain a comparable throughput.

Our approach builds on previous exploratory research on fully-concurrent transactional Java GCs [5]. More precisely, we keep the same strategy of adding the HTM support on the application side, rather than the GC side. Briefly, the initial algorithm required a *Brooks forwarding pointer* [4] and access barriers for load and store operations executed by the mutators. The purpose of the forwarding pointer is to fix the references pointing to an old version of a moved object. It is usually located in the object's header, which we read in our access barriers, immediately after starting a transaction. This way, the pointer is automatically monitored in hardware. If the GC starts moving the object while the object is accessed by the mutator, the transaction will abort and the mutator will retry. Additionally we set a per-object flag when the GC starts moving an object, called *busy-bit*. The barriers check this flag before accessing the object. If set, the mutator will abort and retry; otherwise, the mutator proceeds concurrently with the GC threads. Transactional load barriers are only enabled for volatile reads. This algorithm was only partially implemented, this being sufficient for proving that the transactional barriers had a prohibitively large overhead. The partial implementation mainly consisted of adding the barriers on top of the base GC (CMS), without removing any of the original synchronization. In what follows, we use the same prototype for a feasibility study that will indicate if the new approach is more suitable for a concurrent GC.

We propose *selective access barriers* that would reduce the unacceptable overhead of the previous algorithm to an adequate value. We call them *selective* because they select a particular path depending on the state of the execution. Essentially, the path is chosen based on whether the GC is running or not. First, we define a global flag, called *in-collection*, with the purpose of communicating the state of the GC to the application threads. When the GC becomes active, we set the in-collection flag, wait for each thread to acknowledge the change, then proceed to the collection. On their side, application threads have a check for the value of the flag for each load or store operation. Based on this value, they choose to follow either *the slow path* (featuring transactional barriers) or *the fast path* (normal execution).

However, their acknowledgment when the flag is set is critical for the correctness of the algorithm. If the GC sets the in-collection flag and immediately proceeds to copy the object, it is possible that some of the mutators just passed the aforementioned in-collection check and found the flag unset. In this case they will continue on the fast path. However, the fast path is not suitable for concurrent execution with the GC, since mutator accesses are not protected in any way from interacting with the collection. Thus, this scenario is error-prone and could lead to inconsistent results. In order for all threads to be aware of the modification, they are forced to traverse a short safepoint. The safepoint is requested by the GC before it starts operating on objects. The GC only starts copying the objects after the safepoint ends, when mutators are also allowed to continue their work concurrently. At the time when mutators are released, the in-collection flag is already set and visible for all threads. This measure guarantees that the application threads will take the slow path whenever they work concurrently with the GC.

The check point is not required at the end of the collection, since the only risk there is the possibility of taking the slow path a few extra times by some of the mutators, until all of them see the flag being unset. We consider this potential overhead to be smaller than invoking a safepoint. After all of the application threads "checked in", they can continue running concurrently with the GC using the slow path featuring transactional barriers. Application threads need to check if the flag is set before every load and store access, so that they take the correct path. However, the flag typically stays in the cache of the cores, hence this check does not introduce a significant overhead.

4 BARRIER OVERHEAD EVALUATION

We evaluated the potential improvement of selective barriers over the initial transactional implementation with a series of experiments on a benchmark suite and a client-server system. We enabled both the interpreter and the JIT compiler for our tests. We ran the experiments on an 8-core (16 threads) Intel Core i7-5960X CPU @ 3.00GHz (Haswell), 32GB RAM, with integrated HTM support. In all experiments we ran the original and the modified versions of the CMS collector of Java Hotspot in OpenJDK8.

The main goal of our experiments was to pinpoint the potential performance gain with the optimized version of the transactional GC. More precisely, we measured how many transactions (that are costly) we save in all scenarios. For simplicity, in what follows, we define the transactional overhead as the difference between the execution times of a GC implementation that uses transactional barriers and the original implementation of that GC (the CMS collector in our case). Typically, if a memory access (either load or store) is encapsulated in a transaction, it has a longer execution time. This delay contains the time to start and commit (or worse, abort) a hardware transaction, in addition to the overhead generally associated to operations executed inside transactions. If we sum up all overheads incurred by individual memory accesses, we obtain the transactional overhead associated to an application. In our case, the individual overheads of the transactions we save vary based on many factors, such as the type of the operations encapsulated or conflicts during the execution. In order to have a lower bound for the saved overhead, we settled for the minimum common overhead for all accesses, that of starting and stopping a hardware transaction. Based on this, we aimed to give an intuition of how much of the initial transactional overhead is eliminated by simply removing the measured number of transactions when enabling the selective barriers. Further, we computed an estimation of the actual CPU time of a transactional GC implementation using selective barriers based on our experimental results.

4.1 Transaction Duration

We define the cost of a hardware transaction as the time it takes to start and commit the transaction. The minimum run time of an empty transaction gives us a lower bound for the cost. This helps us estimate a lower bound for the overhead eliminated by the selective barriers; this means that the savings are even more significant in practice than in our experiments.

For this measurement we relied on the CPU's time-stamp counter, through the native `RDTSC` (read time-stamp counter) instruction, which has nanosecond accuracy. In order to have a stable and conclusive result, we repeatedly start and stop an empty transaction in a loop of 10^9 iterations and measure the total time of the loop. From this, we subtract the time it takes to process the loop itself, with no content. The transactional operations (begin and end) are inserted in assembly language, to reduce the overhead of function calls. We pinned the thread to one core and we fixed the frequency of the CPU to the maximum frequency admitted, 3GHz. The results represent the mean over 50 runs. Thus, we consider that a hardware transaction takes in average 29.92 CPU cycles and 9.97 ns (standard deviation < 0.01 ns), on our Intel Haswell machine. We use these values to estimate the savings in terms of execution time provided by the selective access barriers in various scenarios.

4.2 Benchmark Suite

First, we tested our new selective barriers on a series of benchmarks. We chose the DaCapo suite as being one of the most widely used experimental benchmark group. Each benchmark has multiple warm-up iterations and a final iteration, which is the actual run of the benchmark. In a previous exploratory research work [6] we identified a subset of six benchmarks stable in the execution time: *h2*, *tomcat*, *xalan*, *jython*, *pmd* and *luindex*. We configured the benchmarks to execute 10 iterations and to use 8 concurrent threads for their workloads (equal to the number of cores on the machine). All reported results represent the mean over 20 runs.

We measure the total number of transactions that are eliminated by the selective barriers per application execution. However, since all benchmarks execute their workload concurrently on multiple threads, a part of the transactions take place in parallel. If the executions of two transactions are overlapping, then the resulted transactional overhead (with respect to the execution time of the benchmark) is smaller than the sum of the two individual overheads. Thus, the cumulative savings computed over all transactions would be greater than the actual value. In order for the results to be as accurate as possible, we considered using the CPU time as metric,

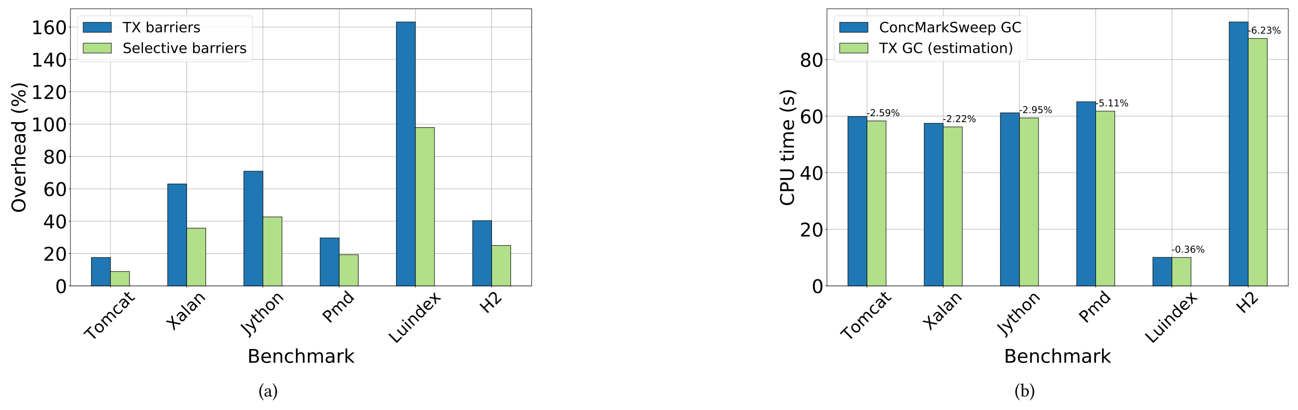


Figure 1: DaCapo: Overhead on top of CMS for simple transactional barriers and selective barriers (upper-bound) (1(a)) and estimated CPU time for a transactional GC with selective barriers compared to CMS CPU time (1(b)).

Table 1: Estimated time savings (DaCapo).

Benchmark	CPU time (s)		Total #tx ($x10^7$)	Min. saved time	
	no tx	tx		(s)	(%)
Tomcat	59.86	70.37	51.94	5.18	7.36
Xalan	57.49	93.72	157.68	15.72	16.77
Jython	61.18	104.55	173.28	17.28	16.52
Pmd	65.1	84.41	67.66	6.75	7.99
Luindex	10.09	26.55	66.02	6.58	24.79
H2	93.28	130.89	143.57	14.31	10.94

rather than the actual wall-clock execution time. The CPU time represents the amount of time (CPU cycles) for which a processing unit was used by the application, excluding all idle times, such as waiting for I/O operations. This measurement is cumulative over all threads, i.e., for a multi-threaded program, the CPU time is expected to be greater than the wall-clock execution time. As the CPU time correctly defines the total execution time over all threads, we can directly associate it with the total number of transactions, as if they were executed sequentially by the application. We retrieve the CPU time for an application with the `perf` Linux tool, using the option `-e cycles`. Thus, we obtained the CPU time for the original implementation using CMS and for the initial transactional implementation, as well as the overhead of the latter over the former. Next, we multiply the measured number of transactions with the transaction cost indicated in Section 4.1, to estimate the time saved with the selective barriers. Table 1 shows the number of transactions and the estimated savings.

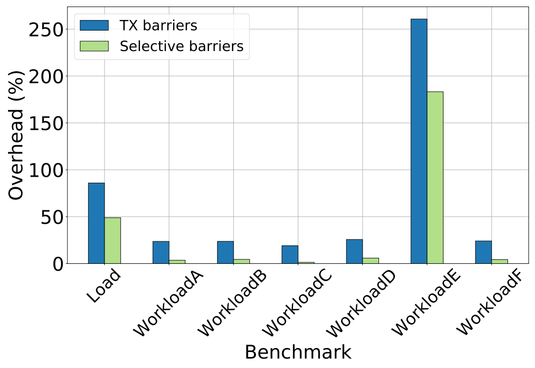
We further employ these results to study the overhead of our selective barriers since the barriers overhead rendered the initial transactional implementation impractical. Figure 1(a) illustrates an estimated upper-bound of the selective barriers overhead, as compared to the transactional overhead of the initial implementation. Given the minimum values used for the overhead computation, e.g., for the transaction cost, the overhead represented in the figure is overestimated, expecting close-to-zero overhead in practice. The purpose of the figure is to show to what extent the overhead is reduced only by introducing a fast-path with no transactional barrier. Basically, despite severely underestimating the overhead savings, the initial overhead is still decreased by up to 50%.

After studying the potential improvement obtained by removing part of the transactions, we looked next at a rough estimation of the CPU time of a fully-concurrent GC with selective barriers. In order to compute this, we needed the CPU time spent in GCs, in addition to the information already gathered. This information is readily available in the details printed by the GC during the execution. Using all the collected data, we estimated the new CPU time according to the following formula:

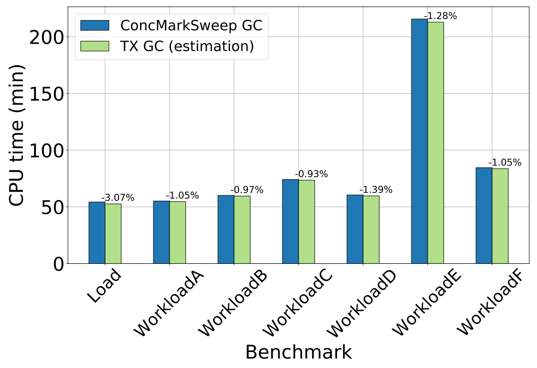
```
est_time = original_time - gc_time + est_gc_time -
                    fast_access_time,
```

where `orig_time` stands for original time, i.e., garbage-collected with CMS; `gc_time` represents the part of the GC that would be made concurrent in our setting, that is, the ParNew collection duration; `est_gc_time` is the sequence that would replace the stop-the-world execution, i.e., the concurrent slow path of the selective barriers, its duration being estimated based on how many accesses could be made during the GC and their cost; finally, we need to subtract the time taken by the accesses that became concurrent, so that we don't count them twice (both on the fast and slow paths), `fast_access_time`.

In a nutshell, this formula translates to: *synthetically replace the GC stop-the-world time with the estimated time on the slow path and avoid counting the slow-path accesses on the fast-path.* We illustrate the result in Figure 1(b). The estimated CPU time of our potential transactional GC with selective barriers is comparable with the original implementation using CMS. The difference is indicated by the percentage labels above the TX GC bars. This result suggests that the performance of a transactional GC enhanced with selective barriers would be on par with that of state-of-the-art GCs, while also eliminating most of the stop-the-world pauses.



(a)



(b)

Figure 2: Cassandra: Overhead on top of CMS for simple transactional barriers and selective barriers (upper-bound) (2(a)) and estimated CPU time for a transactional GC with selective barriers compared to CMS CPU time (2(b)).

Table 2: Estimated time savings (Cassandra).

Benchmark	CPU time (min)		Total #tx ($x10^9$)	Min. saved time	
	no tx	tx		(min)	(%)
Load	54.29	100.94	121.08	20.12	19.93
WorkloadA	55.16	68.21	66.87	11.11	16.29
WorkloadB	60.11	74.35	69.54	11.56	15.54
WorkloadC	74.18	88.31	79.24	13.17	14.91
WorkloadD	60.51	76.08	72.41	12.03	15.82
WorkloadE	215.61	777.87	1006.09	167.18	21.49
WorkloadF	84.54	104.93	100.91	16.77	15.98

4.3 Client-Server Scenario

We also experimented with our selective barriers in a real-life scenario, using the Apache Cassandra 3.9 database server, together with the YCSB client 0.12.0 benchmarks. The benchmarks have two phases: load and run. When *load* is selected, the benchmark simply inserts new records in the database. For the *run* phase, the benchmarks define different combinations of read, update, insert operations as described by Cooper et al. [8]. We configured the benchmarks to execute a fixed number of 10^7 operations on 100 client threads. We considered the results for *load* in general and for the *run* phase of each benchmark separately. For the transactional version, we enabled all barriers on both the interpreter and the JIT. All results represent the mean over 5 runs. Between any two runs of the benchmarks, the server's cache memory is cleaned, the database files are erased and the database is populated again. This is necessary in order to start all benchmarks with the same state and have reproducible experiments.

Table 2 shows the server CPU time with the original CMS implementation, as well as with the non-optimized transactional barriers, total number of transactions that are eliminated due to the selective barriers and a lower bound for the time saved by simply removing the indicated number of transactions.

Figure 2(a) indicates how much we decrease the initial transactional overhead only by removing the cost of starting and committing a transaction for all transactions eliminated by the selective barriers. The gains are not uniform over the benchmarks. We believe this is explained by the type of workload that is executed on the client. For example, WorkloadC is the only read-only benchmark. Since a load operation does not increase the overhead of the transaction significantly, the cost of these accesses is close to our estimation for an empty transaction. Thus, the simple addition of a fast path in the initial algorithm results in at least 93% less overhead for a read-only workload. At the opposite end, the Load and WorkloadE benchmarks are both write-intensive, leading to enormous overheads. The cost of an empty transaction is less fitting for estimating these accesses, resulting in less apparent savings, with a lower bound of 30 – 40%. The rest of the benchmarks, featuring various ratios of read/write operations, reduce the transactional overhead with at least 80%. Except for the write-intensive benchmarks, the upper bound for all overheads with selective barriers is under 5%.

Finally, we apply the same logic as for the DaCapo benchmarks (Section 4.2) to estimate the execution time of a concurrent GC with selective barriers. Figure 2(b) illustrates the comparison between our estimation and the actual CPU time of the Cassandra server configured to use CMS GC. The outcome of this scenario confirms the previous results: a transactional GC with selective barriers can be expected to have a suitable throughput, comparable to real-life collectors, and be almost fully concurrent at the same time.

5 REMOVING GC PAUSES

Given the promising estimations for the performance of the selective access barriers, the next step would be to change the GC safepoint logic to allow the mutators to run at the same time. We further devised a detailed plan of the modifications needed to relax the GC. We specifically aim to make concurrent the young generation collection and its moving phases: (1) copying objects inside the young generation; and (2) promotion to the old generation.

Currently, the GC stops the world during the execution of these operations. Removing the entire safepoint is not straightforward.

We highlight here the most important aspects that need to be addressed and adapted in order to relax the GC:

Root scanning. In most GC algorithms, this particular operation of the garbage collector needs to be executed at a safepoint in order to ensure correctness. The CMS collector, used for the old generation, already has a specific blocking phase for scanning the roots. However, the collector for the young generation, ParNew, used to block the mutators during the whole collection. While there is previous work [13] that designs a lock-free concurrent stack-scanning algorithm, we consider that in our case the complexity of implementing this would surpass the potential benefits. Moreover, based on our experiments the latency of the root scanning operation is negligible. The total CPU time spent in the CMS initial mark phase never exceeds 0.5 seconds for the Cassandra workloads. In comparison, the CPU time spent in young generation collections is around 45 seconds in average for most workloads, going up to over 11 minutes for WorkloadE. Thus, in our design the root scanning represents the only part of the GC for which the mutators are blocked, in both ParNew and CMS.

Tri-color invariant. Most tracing collectors implement Dijkstra's tri-color abstraction. In a nutshell, all objects have to be part of one of the three sets: *white set*, objects that will be collected, *black set*, objects that are scanned and reachable from the roots, will not be collected, and *gray set*, containing objects in the process of having their references scanned. The invariant states that no black object can have a reference to a white object. In our setting, some scenarios may not preserve this invariant. For example, let us assume we have an object A that was already copied (in the black set) and an object B, still in the white set, referenced by an object C in the process of being moved (gray set). If at this point B stops being referenced by C, but A creates a reference to B, object B will be collected even though it is alive. This happens because A's references will not be processed again, while C does not have a reference to B anymore. This scenario is possible because the application is allowed to modify objects concurrently with the moving GC. To ensure that this does not happen in our design, we considered the following solution: whenever a black object receives concurrently a reference that corresponded to a white object, the white object is directly marked as gray.

Stale references. Copying an object means creating new copies for its references as well. However, when mutators run concurrently with the GC threads, other objects that share the same references may not know about the new copies and still see the old references. This potential issue is handled by the forwarding pointer that is already present in the header of the object when using CMS. The busy-bit prevents access to objects that are in the process of being copied; the forwarding pointer ensures that all subsequent accesses to a copied object are written in its primary copy.

Allocation in to-space. This is an optimization that avoids useless copies of an object or extra synchronization when the application allocates new objects concurrently with the GC collection. The young generation is split in three spaces: eden, to-space, from-space. If we allocated the new object in eden as per custom, we would have to copy it to to-space anyway during the collection; if we allocated in from-space, we would have to synchronize with the GC to ensure that the object is collected.

With these considerations in mind, we suggest modifying the current implementation with selective barriers as follows: split the safepoint that protects the entire young generation collection into multiple parts, configured to run at a safepoint or concurrently depending on their task. The root scanning is the first phase and it is run at a safepoint. We expect this pause to be short, as mentioned earlier. On this safepoint we can also piggyback the alteration of the in-collection flag that enables the selective barriers. Then we pass to the next mode of collecting, which will be concurrent. This covers the processing of the references, correctness being ensured by the transactional barriers enabled in the previous step. Finally, since we plan to allocate new objects in to-space, a short safepoint may be necessary for swapping survivor spaces (from-space and to-space). These general guidelines were followed when computing the estimated CPU time for a transactional GC with selective barriers.

6 CONCLUSIONS AND FUTURE WORK

While the computing world moves towards an ever increasing number of processors in commodity hardware and writing correct software becomes overly complex, programming paradigms tend to turn in the direction of lock-free synchronization and automatic memory management. Even though *hardware transactional memory* was already offered for public use in the last few years as a convenient alternative to the classical locking systems, it is yet to be discovered which are its major advantages in practice. We employ HTM with the aim to improve a real-life Java *garbage collector*, by eliminating GC pauses while keeping a comparable throughput. Previous work [5] suggested that current HTM technologies are not yet capable of replacing the existing locking system without introducing a considerable overhead. We reassess the algorithm and propose an efficient optimization, *selective access barriers*. The barriers select the fastest way to proceed depending on the GC state. If a collection is in progress, they install transactional barriers and allow the application to continue concurrently with the GC, while all potentially dangerous accesses are monitored in hardware; otherwise, they do not cause any overhead. We started by implementing the selective barriers on top of the CMS collector. Before fully transforming it into a transactional pauseless GC, we first consider the overhead introduced by our solution when the GC is not executing. We find a lower bound for the possible savings and we show that even in these conditions the initial overhead is reduced by 30 − 40% for write-intensive workloads and by up to 93% for read-intensive workloads. We further estimate the CPU time of a potential transactional GC based on the current data and observe that it is comparable to classical CMS in all cases. This is a good indication that the selective barriers make a full implementation of the pauseless transactional GC worthwhile, as opposed to the former transactional barriers. Therefore, we also make some considerations regarding the challenges encountered when implementing such a GC, and line up the next steps towards its completion. As such, there is a clear direction for future work in this context: the full implementation of a concurrent HTM-based GC and its thorough evaluation in real-life scenarios.

A SELECTIVE BARRIERS: PSEUDO-CODE

Algorithm 1 illustrates the logic of the selective transactional barriers as described in Section 3. It shows an example of a load access barrier. In the pseudo-code oop stands for *ordinary object pointer*, denoting a managed pointer to an object. *Field1* represents an object field that needs to be loaded in the particular case presented in Algorithm 1. The accessed field is loaded in a register called *value*. If the global *in-collection* flag is not set (line 2), the execution takes the fast-path. Otherwise, it uses the transactional barriers. For simplicity, let us assume that the forwarding pointer and the busy-bit represent the metadata of the object. In line 6 we load the metadata of the object of interest, right after starting a transaction (the forwarding pointer is monitored in hardware). We check for the busy-bit as described (line 7). The mutator aborts and retries if the GC manipulates the same object.

Algorithm 1 Selective transactional access-barrier

```
 1: function ACCESS-BARRIER(oop)
 2:     if ¬in-collection then                          ▷ Fast-path
 3:         value ← LOAD-FIELD(oop.field1)
 4:     else                     ▷ Slow-path (concurrent with GC)
 5:         TX-BEGIN
 6:         data ← oop.metadata
 7:         if IS-BUSY(data) then
 8:             TX-ABORT
 9:         end if
10:         value ← LOAD-FIELD(oop.field1)
11:         TX-COMMIT
12:     end if
13: end function
```

The *in-collection* flag is managed by the unique VM thread, which is also responsible for scheduling the GC (Algorithm 2). In a nutshell, the main thread of the VM receives requests for operations in an operations queue and addresses them sequentially in an infinite loop until the end of the application. If a GC is requested (line 3), the VM thread launches a small safepoint to safely set the *in-collection* flag, releases the mutators, then executes the GC operation. Finally, the flag is disabled without any other synchronization (line 8).

Algorithm 2 In-collection flag handling

```
 1: function VMTHREAD::LOOP()
 2:     op ← OperationsQueue.NEXT()
 3:     if op = GC_Operation then
 4:         SAFEPOINT-BEGIN
 5:         in-collection ← 1
 6:         SAFEPOINT-END
 7:         EXECUTE(op)
 8:         in-collection ← 0
 9:     end if
10:     // Other operations
11: end function
```

REFERENCES

[1] 2014. The Apache Cassandra Project. http://cassandra.apache.org. (2014).

[2] Todd A. Anderson, Melissa O'Neill, and John Sarracino. 2015. Chihuahua: A Concurrent, Moving, Garbage Collector using Transactional Memory. In *TRANSACT*.

[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. 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)*. ACM, New York, NY, USA, 169–190. http://doi.acm.org/10.1145/1167473.1167488

[4] Rodney A. Brooks. 1984. 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)*. ACM, New York, NY, USA, 256–262. http://doi.acm.org/10.1145/800055.802042

[5] Maria Carpen-Amarie, Dave Dice, Patrick Marlier, Gaël Thomas, and Pascal Felber. 2015. Evaluating HTM for Pauseless Garbage Collectors in Java. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 3*. IEEE Computer Society, Washington, DC, USA, 1–8. http://dx.doi.org/10.1109/Trustcom.2015.606

[6] Maria Carpen-Amarie, Patrick Marlier, Pascal Felber, and Gaël Thomas. 2015. A Performance Study of Java Garbage Collectors on Multicore Architectures. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '15)*. ACM, New York, NY, USA, 20–29. http://doi.acm.org/10.1145/2712386.2712404

[7] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software Transactional Memory: Why Is It Only a Research Toy? *Queue* 6, 5 (Sept. 2008), 40:46–40:58. http://doi.acm.org/10.1145/1454456.1454466

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. http://doi.acm.org/10.1145/1807128.1807152

[9] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. ACM, New York, NY, USA, 37–48. http://doi.acm.org/10.1145/1029873.1029879

[10] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. 2011. Why STM Can Be More Than a Research Toy. *Commun. ACM* 54, 4 (April 2011), 70–77. http://doi.acm.org/10.1145/1924421.1924440

[11] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*. ACM, New York, NY, USA, 289–300. http://doi.acm.org/10.1145/165123.165164

[12] Balaji Iyengar, Gil Tene, Michael Wolf, and Edward Gehringer. 2012. The Collie: A Wait-free Compacting Collector. In *Proceedings of the 2012 International Symposium on Memory Management (ISMM '12)*. ACM, New York, NY, USA, 85–96. http://doi.acm.org/10.1145/2258996.2259009

[13] Gabriel Kliot, Erez Petrank, and Bjarne Steensgaard. 2009. A Lock-free, Concurrent, and Incremental Stack Scanning for Garbage Collectors. In *Proceedings of the 2009 International Conference on Virtual Execution Environement (VEE '09)*. ACM, New York, NY, USA, 11–20. http://doi.acm.org/10.1145/1508293.1508296

[14] Phil McGachey, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Vijay Menon, Bratin Saha, and Tatiana Shpeisman. 2008. Concurrent GC Leveraging Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. ACM, New York, NY, USA, 217–226. http://doi.acm.org/10.1145/1345206.1345238

[15] Carl G. Ritson, Tomoharu Ugawa, and Richard E. Jones. 2014. Exploring Garbage Collection with Haswell Hardware Transactional Memory. In *Proceedings of the 2014 International Symposium on Memory Management (ISMM '14)*. ACM, New York, NY, USA, 105–115. http://doi.acm.org/10.1145/2602988.2602992