Evaluating HTM for pauseless garbage collectors in Java

Maria Carpen-Amarie*, Dave Dice†, Patrick Marlier*, Gaël Thomas‡ and Pascal Felber*
*Université de Neuchâtel, Switzerland
firstname.lastname@unine.ch
†Oracle Labs, USA
dave.dice@oracle.com
‡Telecom SudParis, France
gael.thomas@telecom-sudparis.eu

***Abstract*—While garbage collectors (GCs) significantly simplify programmers' tasks by transparently handling memory management, they also introduce various overheads and sources of unpredictability. Most importantly, GCs typically block the application while reclaiming free memory, which makes them unfit for environments where responsiveness is crucial, such as real-time systems. There have been several approaches for developing concurrent GCs that can exploit the processing capabilities of multi-core architectures, but at the expense of a synchronization overhead between the application and the collector. In this paper, we investigate a novel approach to implementing pauseless moving garbage collection using hardware transactional memory (HTM). We describe the design of a moving GC algorithm that can operate concurrently with the application threads. We study the overheads resulting from using transactional barriers in the Java virtual machine (JVM) and discuss various optimizations. Our findings show that, while the cost of these barriers can be minimized by carefully restricting them to volatile accesses when executing within the interpreter, the actual performance degradation becomes unacceptably high with the just-in-time compiler. The results tend to indicate that current HTM mechanisms cannot be readily used to implement a pauseless GC in Java that can compete with state-of-the-art concurrent GCs.**

I. Introduction

Today, language runtimes are widely used to execute server applications such as JBoss or Cassandra. A language runtime uses a *garbage collector* (GC) to automatically reclaim memory. For today's servers characterized by large memory requirements and running on multi-core architectures, a GC that suspends the application during an entire collection is not adequate anymore. With a heap size of tens of GB, the pause caused by a single collection cycle can sometimes reach minutes, dramatically degrading server responsiveness [1], [2].

Lately, a new trend started to emerge: the use of *transactional memory* (TM) [3] in memory management systems. TM is a synchronization mechanism designed to make the development of multi-threaded applications easier and less error-prone. It was long debated and argued that software TM (STM) cannot compete with conventional synchronization techniques from a pure performance perspective [4], [5]. However, hardware TM (HTM), recently proposed in consumer-grade processors, appears to be the perfect solution for improving the performance of specialized concurrency problems, such as concurrent garbage collection.

The significant impact of concurrent moving GC on application throughput and responsiveness and the possible solution offered by transactional memory motivate this work. In this paper, we study whether using HTM to synchronize the application and the GC could remove the application pauses while maintaining an acceptable throughput. To that end, we propose a new GC algorithm that uses transactions in read/write barriers to synchronize the application with the GC. When the GC needs to move an object, it marks the object as busy. If an application thread tries to access the object at the same time, the transactional barrier notices the busy state and aborts the transaction. Otherwise, the application threads can work concurrently with the GC threads.

We build on top of an existing GC in one of the most widely used managed runtime environments, the HotSpot Java virtual machine (JVM). We extend Java's *ConcurrentMarkSweep* collector with transactions executed during application reads and writes. We use them to avoid inconsistencies when objects are moved. In this paper we try to answer the question of whether using HTM in GCs is a viable option and we focus on extensively evaluating the performance impact on applications of HTM-based synchronization. We have therefore not fully implemented our concurrent GC algorithm, but only the synchronization between the GC and the application.

We have evaluated our modified version of ConcurrentMarkSweep with the DaCapo benchmark suite on an 8-core Intel "Haswell" machine. Overall, our results show that, in terms of throughput, our GC algorithm would not be able to reach the currently existing throughput-oriented Java GCs. If implemented in the interpreter, HTM in read/write barrier has a reasonable overhead of $\sim$4% after applying some optimizations. However, we observed that the equivalent overhead for the transactional barriers in the *just-in-time* (JIT) compiler is rather large, going up to extreme values of 120%. This result shows that instrumenting read and write barriers with HTMs is today not a satisfactory solution for maintaining a comparable throughput for a concurrent collector. More precisely, our evaluation shows that:

- **Transactional overhead evaluation:** by default, memory access operations made by the application threads are protected by barriers. We encapsulate each operation in transactional barriers. We measure the overhead introduced when we enable in the interpreter individually

the transactional barriers for write, read, and for both operations. We find that transactional write barriers have a negligible overhead, while transactional read barriers' overhead is up to 30%.
- **Transactional barrier optimization:** according to the Java memory model (JMM), only volatile reads need to strictly return the last written value, whereas normal reads are permitted to return any value written in the past. We can use this property to only encapsulate the volatile reads in transactional barriers. In this case, the overhead introduced by the transactional read barrier in the interpreter is less than 4% in most of the tested cases.
- **Transactional overhead in the JIT compiler:** when we evaluate in the JIT compiler the same policy, with transactional read barriers only activated for volatile reads and transactional write barriers always activated, we find that the actual throughput overhead goes well above acceptable limits with values around 20%, 50% and even more than 100% in worst cases.

The rest of the paper is organized as follows. In Section II we define the main concepts used in this work. Our motivation is described in Section III. Our approach is illustrated in Section IV, while Section V shows the experimental results. We discuss related work in Section VI and conclude in Section VII.

II. Background

A. HotSpot Garbage Collectors

The *garbage collector* (GC) is a fundamental component of any Managed Runtime Environment, such as the Java Virtual Machine (JVM). We use the OpenJDK 8 (Open Java Development Kit) implementation, focusing in particular on the GCs of its HotSpot JVM.

All the GCs of HotSpot are tracing collectors. At collection time the collector follows all references, identifying all reachable objects. Non-reachable objects are garbage and the underlying memory is recycled and made available for subsequent allocation requests. Most of the HotSpot garbage collectors are copying collectors. They move the live objects during a collection in order to avoid memory fragmentation. They are also often parallel and span multiple threads to collect the memory in parallel. They are finally often *generational*, following the idea that most allocated objects are not referenced for long (they die young). A generational collector segregates the heap in two generations, a young generation that contains recently allocated objects and an old generation that contains the objects that have survived several collection cycles. A generational collector leverages the observation that young objects tend to die quickly by performing more young collections, which only collect the young generation, than full collections, which collect both generations.

HotSpot defines both stop-the-world and concurrent collectors. Concurrent GCs perform concurrently with the application threads (often called the *mutators*) as much as possible, while stop-the-world GCs suspend the mutators during the whole collection. Even for concurrent GCs, during different phases of the collection, the collector needs to *stop-the-world* in order to avoid concurrency hazards. This is for example the case to identify the references in the execution stacks of the mutators (except in the noteworthy work of Kliot et al. [6]). Stop-the-world pauses of both types of GCs occur at a *safepoint*, i.e., a state in which all the mutators are suspended. There are multiple reasons for safepoints in a program (such as code deoptimization, flushing code cache, biased lock revocation, etc.), but arguably the most frequent cause is represented by garbage collection pauses.

HotSpot proposes 7 different GC algorithms. Out of these, we only consider in our work the 3 most widely used:

- **ParallelOld GC:** generational collector that uses parallel stop-the-world moving algorithms for both generations.
- **ConcurrentMarkSweep GC:** generational collector that uses a parallel stop-the-world moving collector for the *young generation* (called ParNew) and a concurrent low-pause non-moving collector for the *old generation*.
- **Garbage-First (G1) GC:** splits the heap in multiple regions and defines a collection in two phases: a concurrent marking phase and a stop-the-world compacting phase. By controlling the number of compacted regions, G1 has predictable pauses.

B. Transactional Memory

Ever since multi-core systems started to be heavily used, synchronization methods continued to develop, competing for correctness and efficiency. With the rising number of cores per chip, classical synchronization (e.g., using locks) is lagging behind and scalable non-blocking synchronization mechanisms are becoming increasingly more important. A notable example in this direction is represented by *transactional memory* (TM).

First defined by Herlihy and Moss [3], TM became a hot research topic in the last few years. TM avoids the locking mechanism of typical synchronization methods by encapsulating groups of instructions in *transactions*. Any transaction (group of operations) will execute atomically. The application either applies all changes (if the transaction successfully commits), or none of them (if the transaction aborts). The other threads are not able to see intermediate results. Typically, a transaction has two associated logs: a read-set and a write-set. Their purpose is to track all read and write operations, in order to apply them correctly when the transaction commits, or to discard them if the transaction aborts. A transaction aborts if it conflicts with another thread, i.e., if an outside entity tries to modify any of the objects recorded in the read-set or write-set.

The concept of transactional memory covers two main implementations: in *software* (STM) and in *hardware* (HTM). There have been fierce debates whether it is feasible to use Software TM (STM) in real systems or if it is just a research toy [4], [5]. The most important issue of STM is the significant overhead introduced by the instrumentation of the application. Moreover, STM might respect only the notion of *weak atomicity*, i.e., the STM framework cannot detect conflicts between transactional and non-transactional accesses.

Hardware implementation appeared recently [7] and became available to the general public only in mid-2013, with the Intel "Haswell" processor. Even though HTM is faster than STM, it has its own disadvantages. Notably, the size of the transaction is limited and transactions can be aborted by faults, traps, exceptions and specific instructions (debug, I/O, etc.). However, while this requires a non-transactional fallback path,

it also guarantees deadlock-freedom in transactions that would not be able to commit. According to the size limitation, the developer has to carefully plan the contents of a transaction, in order to amortize the cost of starting and committing a transaction. An empty transaction already has a considerable latency. On our test server, with an Intel "Haswell" processor, an empty transaction took 136 cycles. For comparison, a `cmpxchg` (compare-exchange) operation takes 124 cycles and a simple load in the L1 cache has 1-10 cycles.

C. Intel TSX

Intel recently added the *transactional synchronization extensions* (TSX) instruction set to its "Haswell" microprocessor architecture. TSX is a group of instructions that allows the atomic manipulation of memory. They provide two mechanisms of synchronization: *hardware lock elision* (HLE) and *restricted transactional memory* (RTM). The former offers backward compatibility with conventional lock-based mechanisms. Applications using HLE can be used on both legacy and new hardware. In contrast, the latter implements HTM with a new instruction set, which would not run on legacy computers. It allows more flexibility in defining transactional regions; however, it always needs a fallback path in case the transactional execution is not successful. In this work we rely on RTM for our transactional implementation.

RTM contains the following instructions: `XBEGIN`, `XEND`, `XABORT` and `XTEST`. The first two indicate the start and the end of a transactional region. When `XEND` is called, the transaction commits all the changes to the memory. If the transaction is aborted, either by the interaction with another thread or by the explicit `XABORT` instruction, the execution jumps to a fallback handler specified when calling `XBEGIN`. The reason for the abort is saved in the `EAX` register. Finally, the `XTEST` instruction indicates whether the processor is executing a transactional region or not.

III. Motivation

As part of exploratory research[2] to motivate this work, we studied the impact of garbage collection on application performance in two scenarios: *on a benchmark suite* and *on a client-server system*. We considered two main metrics: the application throughput and the duration of GC pauses. We evaluated the GCs featured in OpenJDK 8 HotSpot, while varying the heap size, young generation size and other properties. The experiments were run on a 48-core server with 64 GB RAM. For brevity, we only illustrate the results corresponding to a heap size of 64 GB and young generation size of 12 GB, for both scenarios. We chose the heap equal to the server's memory, while the latter respects Java's guidelines of having around one fourth of the heap size. We chose these dimensions in order to study the behavior of the GCs when the collection space is significantly large. We evaluate the 3 main GCs specified in Section II-A.

A. Throughput evaluation

We first experimented with the DaCapo benchmark suite [8] to evaluate the throughput of the GCs. Each benchmark runs in multiple iterations, out of which the last one is the actual run, whereas the others are marked as warm-up rounds. We identified 7 benchmarks with stable execution time for the final round and/or the total execution: *h2*, *tomcat*, *xalan*, *jython*, *pmd*, *luindex*, and *batik*. Between any two iterations there is a system (full) GC. Figure 1 summarizes our results in this scenario. Since the total execution times have different orders of magnitude, we normalized them with the total execution time of the default Java GC, ParallelOld. The Y axis represents this ratio. The figure clearly shows that G1 performed worse than the default in all cases, while ConcurrentMarkSweep had slightly better results only in 2 cases.

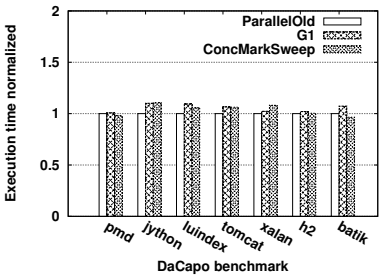

Fig. 1. Normalized execution time for a subset of DaCapo benchmarks running the main Java GCs

This conclusion, that the concurrent GCs had visibly worse throughput than ParallelOld, is also supported by Figure 2. We show the fraction taken by the total pause time in the total execution time of the application for *h2*. The *h2* benchmark mimics an in-memory database and executes a number of transactions against a model of a banking application. In the figure, we call *total pause time* the sum of all the stop-the-world pauses the application encountered during a run.

There are two important observations on the aforementioned figure. First of all, even with a large heap and young generation sizes, the total pause time for a simple benchmark represents a substantial part of the total execution time. We find that, for ConcurrentMarkSweep, the application spent 14% of its execution time by waiting the GC to collect, while for G1 we reach about 14.3%. The second observation is that ParallelOld performs again clearly better than the two other GCs in terms of throughput, with pause time of 6.5% of the total execution time of the benchmark.

Finally, while 14% of the execution time may appear to be a large amount spent in stop-the-world pauses, we can mention as a side note our results for a smaller heap and young generation size. We also tested on a 250MB heap with a 100MB young generation, which represent very low values for today's machines but were still common in 2009 when this DaCapo benchmark was released. In this extreme case, the application spent 63.9% of its execution time in pauses caused by the ConcurrentMarkSweep GC, and 52.9% in the case of G1 GC. As expected, the throughput for ParallelOld is better, pausing only around 28% of its execution time.

B. Responsiveness evaluation

We then used the Apache Cassandra server [9] in combination with the YCSB client [10] to evaluate GCs responsiveness. Cassandra is a distributed on-disk NoSQL database. It provides no single point of failure, and is meant to be scalable and

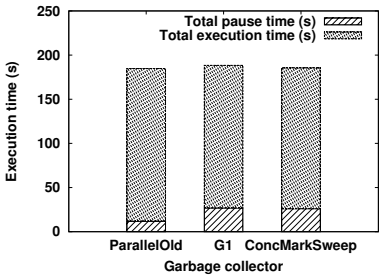

Fig. 2. Fraction occupied by GC pauses in the H2 benchmark execution time for the main Java GCs

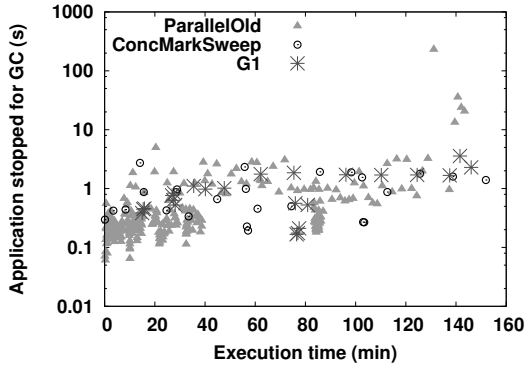

Fig. 3. Application pauses caused by the main Java GCs in Cassandra server

highly available. We used the client in a so-called *loading phase*, which means that it continuously inserted entries into the Cassandra database. We also took advantage of Cassandra's internal data structures and configured it to flush as rarely as possible its records to disk and stress the memory until the server is saturated.

The results are illustrated in Figure 3. The experiment was run for 2 hours; however, the total execution time in the chart (X axis) is longer, accounting also for the time it takes for Cassandra to bring back in memory everything on the disk and replay all transactions. The Y axis represents the duration of the stop-the-world pauses caused by the GC. It is in a logarithmic scale, since some pauses of ParallelOld are with two orders of magnitude longer than for the other two GCs. Thus, the chart shows a pause of up to 4 minutes for ParallelOld, caused by a full collection. There are also several pauses over 10 seconds.

Both G1 and ConcurrentMarkSweep perform better, which shows that concurrency actually helps in improving responsiveness. Nonetheless, their pause lengths go up to 3.5 seconds because of the big heap/young generation size that was continuously kept saturated by the client. These pause times can still be unacceptable on certain systems. For instance, in a distributed environment (such as Cassandra when deployed on multiple nodes) the response time on a node might be critical to keep the system from considering it faulty or crashed. Moreover, Cassandra DB should be able to run without interruption for an indefinite period of time: we only ran for two hours and all 3 GCs finished with pauses bigger than 1s.

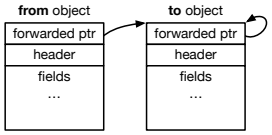

Fig. 4. Brooks forwarding pointers

IV. Our Approach

As discussed in Section III, stop-the-world GCs have high throughput, but they have considerable pauses with large workloads, such as a fully loaded Cassandra server. In order to decrease this pause time while maintaining comparable throughput, we present a novel GC algorithm that should benefit from HTM.

A. Algorithm

Our approach assumes a *Brooks forwarding pointer* [11]. A forwarding pointer normally points to the primary copy of an object (see Figure 4). If the object is not forwarded, then the pointer is self-referential, i.e., it points to itself. The forwarding pointer leads a mutator to the primary copy of an object when the mutator still has a reference to the old copy. This approach allows for lazy updating of the references that point to the old version of a moved object.

In a moving concurrent collector, the application threads work and perform changes at the same time as the concurrent GC threads. This calls for the need to synchronize the access activity of the two groups of threads so as to avoid races. This is done by guarding all mutator accesses (both load and store) with *access barriers*. The barriers are supposed to identify the access to an object in the process of being moved by the GC, wait for the copy to complete, and redirect the mutator to the new location of the object through the forwarding pointer.

In order to ensure consistency, we propose a new GC algorithm having write and read barriers, both of them encapsulated in hardware transactions. We also encode a *busy* bit in the forwarding pointer. Before copying an object, the GC sets the busy bit in the forwarding pointer, which indicates that the mutator has to wait before accessing the object. The GC then copies the object and updates the forwarding pointer to the new location of the object with the busy bit unset, which indicates that the mutator can safely access the primary copy (the new copy of the object).

For the mutator, we have to avoid any interleaving between a copy of an object performed by the GC and a memory access to the same object performed by a mutator. Indeed, in this case, the mutator could access the old version of the object while it is copied, leading either to a read to a stale value, or to a write to the old version of an already copied object. For this reason, before any memory access, the mutator starts a transaction. Then, we identify two possible interleavings:

Case 1: the mutator starts the transaction and the GC starts to copy the object before the end of the transaction. The mutator handles this case by first reading the forwarding pointer, thus adding it to its read set. The address holding the pointer is thus automatically monitored in hardware. If the GC starts to copy

Algorithm 1 Access barrier

```
1: function ACCESS-BARRIER(oop)
2:     TX-BEGIN
3:     f ← oop.fwd
4:     if IS-BUSY(f) then
5:         TX-ABORT
6:     end if
7:     value ← LOAD-FIELD(f.field1)
8:     TX-COMMIT
9: end function
```

the object before the end of the transaction, the GC aborts the transaction because it activates the *busy* bit.

Case 2: the GC starts to copy an object (i.e., activates its *busy* bit) and the mutator starts to access the object before copying ends. The mutator handles this case by testing the *busy* bit immediately after reading the forwarding pointer. If enabled, it means that a copy is in progress since before the memory access, and the mutator thus marks the transaction as aborted.

These steps are illustrated in Algorithm 1. The *oop* in the pseudocode 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 transaction aborts, either in Case 1 or 2, the mutator retries the transaction. We expect aborts to be rare in practice. This approach has two immediate consequences: **(1)** There are no more GC pauses during the moving phase of the GC because the mutators and the GC are only synchronizing through hardware transactions; and **(2)** Transactional operations will add some overhead, which might result in loss of throughput. The overhead comes from the latency of starting and committing a hardware transaction (specifically, the run of the operations XBEGIN and XEND). It is machine specific, but generally comparable to an uncontended atomic `cmpxchg` (atomic compare-and-swap) operation.

B. Implementation

Since HTM is a critical component of our approach, we decided to rely on the abilities of the mainstream Intel "Haswell" processor. As we only want to evaluate the performance overhead caused by the transactions executed for memory accesses, we have not fully implemented our algorithm. For the mutators, we have implemented the barriers, both at the interpreter level and at the JIT level, but we have not modified the CMS algorithm itself: ParNew still stops the world during a young collection, and the old collector is still concurrent without copying the objects. Modifying CMS to fully implement our concurrent moving algorithm would require removing all the GC safepoints and redesigning the initial marking (roots identification) and remarking phases. Hence it would demand a time-consuming implementation whereas our negative results already indicate that it would deliver inferior performance.

While implementing the barriers is relatively straightforward (simply encapsulating each operation in transactions) in practice we had to overcome a few difficulties. By default, HotSpot only injects barriers for store operations (i.e., write barriers). As such, we needed to add barriers for load operations as well. Thus, in order to implement this approach, we focused on the following details:

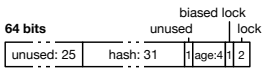

Fig. 5. Bit format for an object header, big endian, most significant bit first

Object's header: In this implementation of Java objects, the forwarding pointer is directly encoded in the header of the object (it is not a separate layer as in the conceptual illustration of the brooks pointers in Figure 4). Since reading the forwarding pointer means fetching the entire object header, we rather modified the header than the forwarding pointer in order to encode the busy state, as required by the algorithm. For this we defined a *busy bit* in the unused area of the object's header (Figure 5). We consider the busy bit as the leftmost bit in the header. An important detail is that the header is encoded differently when using compressed pointers to the objects or a biased locking scheme, leaving no room for the busy bit. Thus, at runtime, we configure Java to run without compressed pointers or biased locking and on 64 bits. We consider the 64-bit setting acceptable, since this is the norm nowadays both for servers and desktop computers. The lack of the other two options, i.e., compressed pointers and biased locking, represents a particular simplified case.

Busy bit encoding: We developed a special encoding of the object's header so that the mutators can identify if the object is being copied or not. When the GC needs to copy an object, it first allocates memory for the new instance, then flips the busy bit and starts the copying process. The bit is removed once the copying is done. Writing the new encoding in the header will cause any other concurrent access from a mutator to abort. The operations *set* and *unset* of the busy bit are executed atomically with a *compare-and-swap* operation.

Write barrier: As previously mentioned, Java HotSpot already featured write barriers by default. We simply wrapped every store operation in a hardware transaction. After the beginning of any transaction, we check for the busy bit and apply Algorithm 1.

Read barrier: We added the read barrier to the implementation. We encapsulated the load operation in transactions and replaced the original load with the new transactional load.

Read barrier optimization: Due to the important overhead introduced by the insertion of read barriers (see Section V), our next goal was to optimize their implementation. We carefully studied the Java memory model (JMM) and observed that only the *volatile* load operations must return exactly the last value written, while the normal loads can return any value previously written. We added a check for volatile loads. Then we only called the transactional load for volatile reads. This greatly reduced the overhead while still complying with the JMM.

V. Experimental Results

In order to test our algorithm and to compare it to the currently existing GCs in OpenJDK, we used an Intel Core i7-4770 CPU at 3.40GHz (codenamed "Haswell") with fully integrated HTM support. We experimented with the DaCapo benchmark suite. Specifically, we ran the subset of benchmarks that we discovered to be stable (see Section III). All selected benchmarks had small variations (less than 5% in most cases)

TABLE I. INDEPENDENT OVERHEAD FOR TRANSACTIONAL READ AND WRITE BARRIERS (INTERPRETER)

Txnal barriers	None	Write	Read	Read/Write
Duration(s)	96.09	98.65	108.38	114.65
Tx-cycles(%)	0.00	3.17	20.21	21.92
Tx-aborts(%)	0.00	0.00	0.02	0.04
Time Overhead(%)	-	2.6	11.3	16.2

TABLE II. TRANSACTIONAL CYCLES AND ABORTS IN THE INITIAL IMPLEMENTATION AND AFTER THE OPTIMIZATION (INTERPRETER)

Benchmark	All read-TX		Volatile read-TX	
	TX-cycles(%)	Aborts(%)	TX-cycles(%)	Aborts(%)
H2	33.82	0.01	7.06	0
Tomcat	19.26	0.04	2.72	0.01
Xalan	19.6	0.06	2.55	0.01
Jython	21.12	0.07	9.77	0
Pmd	18.21	0.05	5.08	0.01
Luindex	27.8	0.01	8.58	0

for both the total execution time and for the final round. One exception is represented by the *batik* benchmark, which is only stable for the total execution time, varying with more than 11% for the final iteration. Since we only focused on the execution time of the actual run of the benchmark (the final round), we removed that application from our subset.

We started by measuring the overhead introduced by the read/write barriers and the execution time in interpreter mode of a single run of each benchmark with the new implementation. We gathered statistics using the `perf` Linux tool. We measured the overhead in 4 situations:

- when the transactional barriers are disabled (the default access barriers are in place);
- when only the write transactional barrier is enabled (there is no read barrier, same as in the default implementation);
- when only the read transactional barrier is enabled; and
- when all transactional barriers are enabled.

Table I illustrates the overhead of the read/write barriers for the *jython* benchmark. All other benchmarks have similar overheads, thus we only exemplify on this complex and stable benchmark. First of all, we observe that the write barrier has an almost negligible effect on the performance of the application in interpreter mode. However, the read-barrier raises the overhead to an unacceptable level. We also notice that the proportion of transactional aborts is close to 0. This indicates that the overhead comes from the implementation of the algorithm itself, rather than the amount of aborts.

As a consequence of the obtained statistics, partly illustrated in Table I, it became obvious that the considerable overhead of the newly-introduced read barriers made such an implementation unpractical for a real-life GC. Even though the currently existing GCs could have serious issues regarding the pause times, a garbage collector with such overhead would always perform worse than the classic GCs.

We then proceeded to optimize the implementation, with the aim of decreasing the read barrier overhead in the limits allowed by the proposed algorithm. As mentioned in Section IV, we focused our attention on the Java memory model (JMM) and further improved our implementation to

TABLE III. TRANSACTIONAL BARRIERS OVERHEAD IN THE INITIAL IMPLEMENTATION AND AFTER THE OPTIMIZATION (INTERPRETER)

Benchmark	No TX (ms)	All read-TX		Volatile read-TX	
		Exec time(ms)	Overhead(%)	Exec time(ms)	Overhead(%)
H2	78675	103212.6	31	81550.3	3.6
Tomcat	8395.6	9727	15.8	8591.3	2.3
Xalan	10069.3	11586.3	15.06	10119	0.5
Jython	86936	105447.6	21.29	90007	3.5
Pmd	9274.6	10863.6	17.13	9550	2.9
Luindex	10933	13823.3	26.4	11810	8

TABLE IV. TRANSACTIONAL BARRIERS OVERHEAD IN THE JIT

Benchmark	Execution time (ms)		Overhead (%)
	no Tx	Tx	
H2	4681	5774.3	23.35
Tomcat	2698.6	2966	9.9
Xalan	1172.6	1443.6	23.11
Jython	3606	5228	44.98
Pmd	3967.6	4145	4.47
Luindex	559.6	1248.3	123.07

only enable the transactional barriers for volatile reads. We tested the new implementation on the aforementioned subset of DaCapo benchmarks. In interpreter mode, the implementation for the write-barriers did not change, hence, its small overhead stays the same. Thus, in all the following experiments, we have enabled transactional barriers both for volatile reads and for all writes, in order to check the total overhead of the algorithm.

First, we studied the percentage of transactional cycles and aborts for the two strategies in interpreter mode: transactions for all reads and transactions only for volatile reads. We observe in Table II that the latter already significantly decreases the number of transactional cycles, which goes up to 10% (for *jython*), as opposed to up to 33% for the former (*h2* benchmark). Moreover, even though the percentage of transactional aborts is negligible in all cases, the second strategy has almost no aborts for all benchmarks.

Table III presents the execution times and the overhead in interpreter mode, over the original implementation for the two implemented versions: transactions for all reads and for volatile reads only. All execution times are computed as the average over three consecutive runs. The table clearly shows the difference between the two mentioned strategies: when only the volatile reads are encapsulated in transactions the overhead is less than 4% in 5 cases out of 6 (with the exception of *luindex* with an overhead of 8%). The other strategy results in overheads of at least 15%, going up to 30%. The results for the volatile optimization in Table III show a pronounced improvement over the initial implementation of the algorithm.

Finally, we checked the cost of transactional barriers when implemented in the JIT compiler, instead of the interpreter. We inserted them for all volatile loads and all store operations. As such, we were able to compare the relative overhead of our implementation in the interpreter with the results for the JIT implementation. Table IV lists the execution times and the overhead for our subset of DaCapo benchmarks. The benchmarks were configured to run for 5 iterations, and the execution time represents the value of the last iteration (the actual run). The execution time in the table represents the

average over 3 consecutive runs. We found that the overhead for transactional barriers implemented in JIT is significantly larger than in the interpreter. This happens because the interpreter is much slower than the compiled code, but the overhead introduced by the transactional barriers is independent of the JIT. Thus, the same overhead would seem smaller on the interpreter. Our results indicate that the real overhead of the transactional barriers is over 4% in all cases, going up to 123% for the *luindex* benchmark. In conclusion, even optimized, the algorithm has an unacceptable throughput overhead.

VI. Related Work

Nowadays, with HTM being a part of mainstream processors, transactional memory becomes an interesting and efficient alternative for classical synchronization mechanisms such as locks and barriers. One important field that could greatly benefit from HTM is the implementation of garbage collectors. As a consequence, several studies and proposals have been conducted in this direction.

Overall, there has been much research on real-time garbage collectors over the last few years [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22]. An early effort in implementing garbage collectors with transactional memory was made by McGachey et al. [23]. They used STM to implement a new concurrent GC designed to enforce an under 1 ms pause for 90% of collections, under 10 ms for 90% of the rest and under 100 ms for what is left. They successfully reached their goals, thus making a first step towards combining TM with GC.

Once HTM has been supported in some specialized processors and large systems (e.g., Azul, Sun), various proposals have studied the potential benefits of HTM for concurrent scenarios [7], [24]. A particularly interesting work in this direction has been proposed by Iyengar et al. [25], who implemented an HTM-based Java GC on a specialized multi-core CPU build by Azul systems. The *Collie* algorithm represents a first contribution in this field. Its evaluation shows that, thanks to the use of the HTM, Collie has a better responsiveness and throughput than C4 [21], their baseline copying and concurrent collector without HTM. However, the paper does not compare Collie with a throughput-oriented collector such as ParallelScavenge in HotSpot, and it is thus difficult to know whether using HTM in a concurrent collector can help to achieve high throughput.

In mid-2013, Intel made available to public their HTM implementation in the "Haswell" CPU. This created a widely available platform for studying and applying HTM to concurrency problems. Alistarh et al. [26] leveraged HTM for adding automatic memory reclamation for typical C/C++ data structures, such as hash table, linked list, non-blocking queue, etc. According to the evaluation, the performance of the StackTrack algorithm is superior to other memory reclamation mechanisms. However, it can also degrade the throughput of the data structure by up to 50%.

Ritson et al. [27] conducted a performance study on three main GC algorithms of the Jikes research virtual machine (RVM), implemented with HTM. They assess the benefits of using transactions for parallel semispace copying collection, concurrent replicating GC and parallel bitmap marking. They find that the first algorithm is significantly improved by HTM (48-101%), while the other two do not have any benefits, nor drawbacks. They explain that it is difficult to plan the work of the GC such that the cost of the transactions is fully amortized. We consider that our contributions complement this work, with the study of the impact of HTM on *real-life* Java garbage collectors and the proposal of the novel HTM-based algorithm.

VII. Conclusion and Future Work

Multi-core processors are omnipresent nowadays and the number of cores per CPU is expected to grow further in the coming years. Even though garbage collectors have started to adapt to this trend, they are still lagging behind and do not take full advantage of the system's parallelism. The most widely used GCs in OpenJDK's HotSpot VM—ParallelOld, ConcurrentMarkSweep, and G1—are all parallel, but none of them is fully concurrent. In other words, all of them use a different thread for collecting but, at the same time, they all need to stop the application threads at different phases of their collection. Moreover, as mentioned in Section III, the pauses caused by the GCs are significant: up to a few seconds for the most efficient GCs and even a few minutes for ParallelOld.

These limitations led us to consider creating a concurrent GC algorithm that employs hardware transactional memory. Not only would this remove main stop-the-world pauses, but it would permit the GC to take full advantage of a multi-core system without draining its resources. Because of the fixed costs in the stop-the-world operations, it is fairly common to greatly over-provision the heap, i.e., to set the heap size several times larger than the expected size of live reachable data. Thus, a pauseless copying collector would also allow us to avoid heap over-provisioning, besides removing the pauses.

There are two different approaches for implementing HTM inside the GC: one could either instrument the GC or the application accesses. In our algorithm we follow the latter approach, and replace the blocking access barriers of the mutators with transactions. We built our implementation on top of HotSpot's ConcurrentMarkSweep collector. Before embarking on fully implementing a new GC, we focused on assessing the overhead introduced by our transactional implementation, with the goal of verifying if the approach is feasible or not. We tested on a widely-used benchmark suite, DaCapo, using the subset of benchmarks that are the most stable in terms of execution time. We implemented the transactional barriers in the interpreter and in the JIT compiler.

We found that in intrepreter mode, after optimzing the algorithm, we obtain a reasonable overhead of $\sim$4% in most cases. However, by checking the equivalent overhead values for transactional barriers in the JIT compiler, we find that it is not within acceptable limits, having 4 out 6 benchmarks with overheads greater than 20%, going as high as 123% for one benchmark. For this reason we do not consider that the full implementation of our transactional pauseless GC would bring benefits for real applications.

A feasible direction for further research would be to follow the other type of transactional implementation and instrument directly the GC code, e.g., by implementing and improving the Collie algorithm with Intel's TSX. Another option would be to head for a different type of GCs, such as those based on reference-counting. At a first sight, they offer promising opportunities for improvement with HTM.

REFERENCES

[1] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro, "Assessing the scalability of garbage collectors on many cores," *Best papers from PLOS'11, SIGOPS Operating System Review (OSR)*, vol. 45, no. 3, pp. 15–19, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/2094091.2094096

[2] M. Carpen-Amarie, P. Marlier, P. Felber, and G. Thomas, "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*, ser. PMAM '15. New York, NY, USA: ACM, 2015, pp. 20–29. [Online]. Available: http://doi.acm.org/10.1145/2712386.2712404

[3] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 289–300. [Online]. Available: http://doi.acm.org/10.1145/165123.165164

[4] 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*, vol. 6, no. 5, pp. 40:46–40:58, Sep. 2008. [Online]. Available: http://doi.acm.org/10.1145/1454456.1454466

[5] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui, "Why stm can be more than a research toy," *Commun. ACM*, vol. 54, no. 4, pp. 70–77, Apr. 2011. [Online]. Available: http://doi.acm.org/10.1145/1924421.1924440

[6] G. Kliot, E. Petrank, and B. Steensgaard, "A lock-free, concurrent, and incremental stack scanning for garbage collectors," in *Proceedings of the 2009 International Conference on Virtual Execution Environement*, ser. VEE '09. New York, NY, USA: ACM, 2009, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/1508293.1508296

[7] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 157–168. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508263

[8] 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*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 169–190. [Online]. Available: http://doi.acm.org/10.1145/1167473.1167488

[9] Apache, "The apache cassandra project," http://cassandra.apache.org/, 2014.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152

[11] R. A. Brooks, "Trading data space for reduced time and code space in real-time garbage collection on stock hardware," in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, ser. LFP '84. New York, NY, USA: ACM, 1984, pp. 256–262. [Online]. Available: http://doi.acm.org/10.1145/800055.802042

[12] D. Detlefs, C. Flood, S. Heller, and T. Printezis, "Garbage-first garbage collection," in *Proceedings of the 4th International Symposium on Memory Management*, ser. ISMM '04. New York, NY, USA: ACM, 2004, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/1029873.1029879

[13] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard, "Stopless: A real-time garbage collector for multiprocessors," in *Proceedings of the 6th International Symposium on Memory Management*, ser. ISMM '07. New York, NY, USA: ACM, 2007, pp. 159–172. [Online]. Available: http://doi.acm.org/10.1145/1296907.1296927

[14] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek, "Schism: fragmentation-tolerant real-time garbage collection," in *Proceedings of the 2010 International Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 146–159. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806615

[15] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones, "Parallel generational-copying garbage collection with a block-structured heap," in *Proceedings of the 2008 International Symposium on Memory Management*, ser. ISMM '08. New York, NY, USA: ACM, 2008, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/1375634.1375637

[16] D. Doligez and X. Leroy, "A concurrent, generational garbage collector for a multithreaded implementation of ml," in *Proceedings of the 1993 International Symposium on Principles of Programming Languages*, ser. POPL '93. New York, NY, USA: ACM, 1993, pp. 113–123. [Online]. Available: http://doi.acm.org/10.1145/158511.158611

[17] T. A. Anderson, "Optimizations in a private nursery-based garbage collector," in *Proceedings of the 2010 International Symposium on Memory Management*, ser. ISMM '10. New York, NY, USA: ACM, 2010, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/1806651.1806655

[18] S. Marlow and S. Peyton Jones, "Multicore garbage collection with local heaps," in *Proceedings of the 2011 International Symposium on Memory Management*, ser. ISMM '11. New York, NY, USA: ACM, 2011, pp. 21–32. [Online]. Available: http://doi.acm.org/10.1145/1993478.1993482

[19] B. Steensgaard, "Thread-specific heaps for multi-threaded programs," in *Proceedings of the 2000 International Symposium on Memory Management*, ser. ISMM '00. New York, NY, USA: ACM, 2000, pp. 18–24. [Online]. Available: http://doi.acm.org/10.1145/362422.362432

[20] K. Sivaramakrishnan, L. Ziarek, and S. Jagannathan, "Eliminating read barriers through procrastination and cleanliness," in *Proceedings of the 2012 International Symposium on Memory Management*, ser. ISMM '12. New York, NY, USA: ACM, 2012, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/2258996.2259005

[21] G. Tene, B. Iyengar, and M. Wolf, "C4: The continuously concurrent compacting collector," in *Proceedings of the 2011 International Symposium on Memory Management*, ser. ISMM '11. New York, NY, USA: ACM, 2011, pp. 79–88. [Online]. Available: http://doi.acm.org/10.1145/1993478.1993491

[22] C. Click, G. Tene, and M. Wolf, "The pauseless gc algorithm," in *Proceedings of the 2005 International Conference on Virtual Execution Environement*, ser. VEE '05. New York, NY, USA: ACM, 2005, pp. 45–56. [Online]. Available: http://doi.acm.org/10.1145/1064979.1064988

[23] 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*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 217–226. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345238

[24] A. Dragojevic, M. Herlihy, Y. Lev, and M. Moir, "On the power of hardware transactional memory to simplify memory management," in *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ser. PODC '11. New York, NY, USA: ACM, 2011, pp. 99–108. [Online]. Available: http://doi.acm.org/10.1145/1993806.1993821

[25] 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*, ser. ISMM '12. New York, NY, USA: ACM, 2012, pp. 85–96. [Online]. Available: http://doi.acm.org/10.1145/2258996.2259009

[26] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit, "Stacktrack: An automated transactional approach to concurrent memory reclamation," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014, pp. 25:1–25:14. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592808

[27] 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*, ser. ISMM '14. New York, NY, USA: ACM, 2014, pp. 105–115. [Online]. Available: http://doi.acm.org/10.1145/2602988.2602992