Hardware Read-Write Lock Elision on Intel Processors

Tiago João dos Santos Lopes

Instituto Superior Técnico, University of Lisbon

Abstract. *Transactional Memory* (TM) is a promising alternative to lock- based synchronization mechanisms. This report analyses the state of art and existing implementations of TM, focusing, in particular, on *Hardware Read-Write Lock Elision* (HRWLE). HRWLE is a recently proposed technique that exploits the hardware TM supports of IBM Power8 (P8) processor to build a speculative implementation of the well-known read-write lock abstraction. Unfortunately, though, the reliance on hardware features that are only available on P8 represents a key limitation of HRWLE, which prevents its usage on commodity processors by other vendors - in particular Intels CPUs, which also ship with *Hardware Transactional Memory* (HTM) support and are more commonly employed than P8.

The dissertation proposal detailed in this document will aim at tackling precisely this limitation, by designing, implementing and evaluating alternative HRWLE algorithms that will not rely on specific features of Power8 processors and, as such, will be usable in broader contexts.

In addition to defining goals, this report also presents an initial algorithm implementation and some preliminary results compared to other common TM implementations.

Keywords: transactional memory, concurrency control, read-write lock, lock elision, hardware

Table of Contents

1 Introduction

During various decades processors frequencies have been enjoying an exponential increase. Since early 2000, though, this trend stopped as manufacturers hit the so called "Power Wall": due to thermal issues, it is nowadays economically infeasible to further increase the operational frequencies of single core processors. This has brought a paradigm shift not only in the way hardware is designed, turning multi-core processors into a mainstream technology, but also in the way software is built - bringing parallel computing to the forefront of software development.

Unfortunately, developing parallel applications is well known to be a challenging task. One major source of parallel applications complexity is the implementation of a synchronized access to shared resources. Indeed, the classic approach to synchronization problems is to rely on a lock-based scheme, which are known as susceptible to several problems such as deadlocks, livelocks, priority inversions, etc. Given the increased relevance of parallel computing, over the last decade a large research effort has been devoted to identifying simpler, yet highly efficient, alternative synchronization paradigms.

TM is probably one of the alternative synchronization methods to have been most intensively investigated as of late. Making use of Database Systems concept of Transactions, TM is an *Automatic Mutual Exclusion* method, where Programmers no longer need to worry with the synchronization, needing only to code which operations to execute concurrently. TM system would then ensure atomicity by detecting and resolving any conflict arising between concurrent transactions. The TM abstraction can be implemented in software, known as *Software Transactional Memory* (STM), hardware (HTM) or combinations thereof.

Compared to other synchronization methods, HTM focuses on minimizing transaction overhead through hardware support, reducing or even removing the need of programming instrumentation on read and write accesses to shared resources. Various studies [13,16,17,26,27,31] have clearly shown that HTM can achieve, at least in certain workloads, impressive performance gains when compared to software based implementations. Unfortunately, though, existing HTM implementations also suffer of several relevant restrictions that can severely hamper its performance.

Indeed, even though existing HTM implementations come in different flavors depending on [1,12,26,32], they all share a key common treat: they all support transactions that perform a limited number of memory accesses. Whenever a transaction exceeds the maximum HTM capacity, it needs to be executed using a fall-back synchronization method, namely a *Single Global Lock* (SGL) that executes pessimistically and whose activation causes the immediate abort of any concurrent HTM transaction.

HRWLE [15] is a recently proposed synchronization technique that builds on HTM and aims to overcome the above mentioned limitation of HTM.

HRWLE exposes the API of a classic read-write lock, which is executed in a speculative fashion using, in an original way, the hardware supports provided

by the IBMs P8 HTM implementation, i.e., allowing for the safe concurrent execution of readers and writers.

HRWLE functions on the concept that HTM works detecting any data conflict of other transactions and operations on its own atomic sequences and on the concept that the system knows if the transaction will be read-only or will implement changes to the data.

By leveraging these two concepts, HRWLE runs read-only operations outside the scope of hardware transactions, which frees them from any capacity constraint. To guarantee atomicity and snapshot consistency it makes use of *POWER8* (P8)s *suspend and resume* feature, which allows readers to communicate with writers and notify them about their existence. This allows writers to postpone their commit request until any concurrent reader has completed its execution. This quiescence mechanism, combined with the atomicity guarantees provided by HTM transactions (used by writers), is sufficient to isolate readers from any concurrent writer and preserve the original correctness semantics.

Although this method has a good performance, it is limited to processors with *suspend and resume* feature, that is, to P8 processors. The need for a *suspend and resume* feature in HRWLE is due to the fact that HTM prevents any form of communication between concurrent transactions. In fact, any attempt to read by a hardware transaction any memory position that is concurrently updated (by a reader, running in a non-transactional context) is automatically detected as a conflict by HTM and causes the abort of the hardware transaction. Conversely, by suspending the hardware transaction that encompasses the writer logic, the communication (via shared memory) between readers and writes is performed outside of the scope of the hardware transaction. Hence, it allows writers to be informed by readers (via shared memory) about their execution state, without causing writers aborts.

As such this work purpose the development of an algorithm with the same concept as HRWLE, namely supporting the concurrent execution of un-instrumented readers as well of writers using HTM, without however requiring the use of the *suspend and resume* mechanism, which is presently a unique architectural feature of Power8 processors that is not supported by other HTM implementations. Achieving this goal would therefore greatly amplify the generality of the HRWLE approach, enabling its use also in mainstream commodity processors by Intel - whose HTM implementation, referred to as *Transactional Synchronization Extensions* (TSX), does not support the *suspend and resume* mechanism.

This document is divided in 3 parts: Chapter 2 describes the current state of art in Transactional Memory and Read Write Lock Implementations. Chapter 3 overviews the proposed approach, reports some promising preliminary results, and defines the work plan for the remainder of this dissertation. In Chapter 4 we present the conclusions of this report.

2 State of Art

With the emergence of multi-core architectures, the need for a synchronization method between parallel threads accessing shared resources has been a critical priority. In fact, the conventional synchronization approach based on locking is well known to suffer from several problems.

Coarse-grained locking, although easy to implement, is far too pessimistic as it can overly restrict parallelism, failing to take full advantage of modern multi-core systems. Fine-grained locking, although enabling good performance, is complex to implement correctly, debug and reason about [2]. Furthermore, it compromises a key desirable property of software: composability [2]. Using Locks as a synchronization method not only delays the access to critical sections due to the lock, effectively disabling concurrent access to such values, but also delays the threads themselves with its additional overhead during its normal workloads.

Transaction, as a concept, was first developed for databases, as a set of operations that manipulate data atomically. The main purpose was to keep the database consistent, while allowing the concurrent access to the database. To achieve this, transactions have to be Atomic, Consistent, Isolated and Durable (ACID).

These features are also essential for parallel programming. Atomicity ensures that changes done within a transaction appear as all or none to other code. Consistency ensures that data is always in a consistent state. Isolation is relevant since all changes done from within a transaction must remain invisible to all other transactions. Durable as in case of failure the system can either recover entirely or discard the changes by this set of instructions.

2.1 Transactional Memory

TM borrows the abstraction of transactions from databases to the parallel programming domain. It provides programmers with the ability to execute transactions on shared memory data. These transactions are either committed by TM (i.e., the changes of the transaction are applied atomically to the data) or aborted (i.e., the changes are discarded as if they never happened).

TM is a parallel programming paradigm that avoids the pitfalls of traditional locking techniques while promising the performance of fine-grained locking [2]. Programmers using TM need only to worry on their applications logic, not on how to implement synchronization, thus easing the development of concurrent applications that are both scalable and thread-safe in parallel computing.

TM algorithms can be classified according to *data versioning*, *conflict detection*, *granularity* and *read visibility*.

- *Data versioning* has the objective of guaranteeing consistency among all reads and writes. It is implemented in several different methods with its objective being to guarantee all transactions work on a consistent snapshot of the systems memory. TM are mainly divided into *eager versioning* and *lazy versioning*.

- *lazy versioning* stores all memory changes the transaction implements in a buffer to insert in the shared memory on commit. If the transaction is successful the new values are copied to the memory, which results in a small delay as all values are copied. If however the transaction is aborted no further operations are necessary as the values were never written in the system.
- In *eager versioning* however the transaction writes its new values directly in memory, storing the old value in a log for its possible abort. This allows its commit to be much faster, however if the transaction is aborted it must recover all overwritten values causing some additional delay in conflicting transactions.

– *Conflict Detection* is needed when two of more transactions access the same value and at least one of them changes the value before all other transactions sharing access commit. To resolve these situations all reads and writes are tracked and checked for collisions in one of two ways: *pessimistic conflict detection* and *optimistic conflict detection*.

- With *pessimistic conflict detection* the system regularly checks the transactions accessed values. This allows for a early and quick conflict detection at the cost of performance due to its constant checks.
- *optimistic conflict detection* assumes a conflict will not occur in a transaction, checking all values before commit, thus avoiding the performance drop of constant conflict checking. However this detection has the downside that conflicts are only detected at the very end, possibly delaying the abortion for a long time and wasting resources in an aborting transaction.

– *Granularity* is the level at which the TM detects conflicts. *Granularity* is generally either *word-based*, *object-based*, *value-based* or *cache-line based*. *Word-based* granularity means that the TM system detects conflicts between 4 or 8 bytes. *Object-based*, as the name implies, means the system only checks each objects atomicity, leading to possible false conflicts of different variables inside an object. *Value-based* on access locally stores read address and value, allowing the transaction to later confirm the new and previous value are the same. Finally *cache-line based* is a *hardware* specific granularity explained further on.

– *Read Visibility* can be divided in *visible*, where reader inform which memories they have accessed increasing memory checks but enabling early conflict detection, and *invisible*, where readers do not inform other active transactions, forcing writers to check if reads and writes are complete and a consistent snapshot of the system is maintained on commit.

2.2 Software Transactional Memory

Due to the difficulty of manufacturing and testing hardware based TM solutions, STM was developed to implement TM only as a software framework, enabling portability across different hardwares.

STM relies on instrumented read and write accesses to shared memory locations from transactional blocks. This instrumentation then allows the software to detect conflicts through *data versioning* and *conflict detection* as previously mentioned. This generates a higher overhead compared to the *hardware-based* alternative. On the other hand one of its main advantages is the transaction size it can support, unlike *hardware-based* solutions.

Due to the low cost and high flexibility of software implementations, many different designs of STM were developed. STM can be divided according to the previous categories. Table 1 shows some popular and efficient STM implementations.

Table 1: Popular STM characteristics

	Data Versioning	Conflict Detection	Granularity
TL2 [10]	Lazy	Optimistic	Word/Object
TinySTM [14]	Lazy/Eager	Pessimistic	Word
NORec [9]	Lazy	Pessimistic	Value

Transactional Locking II *Transactional Locking II* (TL2), proposed by Dave Dice et al. [10], works as a *two-phase locking scheme*, maintaining a *global version clock*, which is incremented by all writing transactions, and *versioned write-locks* for every shared memory location. It works with *optimistic conflict detection* and *lazy versioning*.

On start all transactions read and store the current *global version clock* in a local variable to identify its *read-version number*. The transaction then runs the user transactional code locally, maintaining a list of *versioned write-locks* of all read values (read-set) and written values (write-set). The transaction also verifies in each read value that its current version is $\leq$ *read-version number* and the read values lock is free to guarantee that the value has not been modified since the transaction began. When it finishes the writer acquires the write-set locks using a bounded spinning (aborting after a fixed period of unsuccessfully acquiring a lock). It then performs a increment-and-fetch operation of the *global version clock* recording its value in a local write-version variable. Finally it revalidates the read-set $\leq$ *read-version number* to guarantee no accessed memory locations were modified during the transaction. If in both checks a value is locked or its value does not comply to the rules above then the transaction aborts.

TinySTM Pascal et al. later proposed *TinySTM* [14], a word-based variant of LSA [29]. *TinySTM*, like TL2 uses both *global version clock* for snapshot consistency and *versioned write-locks* for shared memory addresses. However, instead of locking all needed writes just before commit the algorithm acquires locks on read. *TinySTM* works with *pessimistic conflict detection* and is presented as able to use both *versioning* methods.

Read-only transactions are benefited in this algorithm, the reader verifies the shared memories lock is free, reads the corresponding value and then checks the lock again to confirm that no changes occurred in the meantime. A reader may need to extend its snapshot in case it is reading a value that has a version number greater than the transactions. This is done by validating the read-set and making sure they have not been updated meanwhile.

Write transactions acquire the lock to guarantee that there are no concurrent writers. If the lock bit is set the writer verifies its the current lock owner, otherwise waits or aborts. In the presented *TinySTM* transactions are set to abort immediately. This is useful in workloads with high contention as it minimizes the amount of useless work done.

TinySTM, as mentioned above, can use both *eager versioning*, with *write-through*, resulting in a smaller overhead and delay for other transactions on successful commit, or *lazy versioning* with *write-back*, resulting in a larger overhead in all transactions but smaller delay on abort.

It also presents the concept of *Hierarchical locking*, a strategy to reduce the validation cost of read-sets by reducing the number of read locks while avoiding the increase of aborts due to shared memory with the same lock. *Hierarchical locking* is specially useful if transactions read many memory locations and there are few competing write transactions.

NOrec *No Ownership records* (NOrec) presented by Luke Dalessandro et al. [9] is a highly scalable STM on read-mostly workloads, allowing any reader to promote into a writer at anytime limiting however the algorithm to a single write transaction system-wide. NOrec uses *lazy versioning* and *pessimistic conflict detection*.

NOrec minimizes its overhead by using *Transactional Mutex Lock* (TML), a global clock counter, which allows writers to be serialized. By using TML readers only store a snapshot of the TML and a read-set, consisting of both read values and their addresses. On commit the reader checks its stored TML value and current TML value. If the value is the same then it finishes committing successfully. If the value is different, the reader needs to perform a validation of its read-set, checking its stored reads and current values to confirm its read-set is consistent.

Writer transactions buffer all their writes into a log, attempting to acquire the lock only on commit. This reduces the time TML is held by a transaction, allowing read-only transactions to commit more easily.

2.3 Hardware Transactional Memory

TM was initially proposed as a hardware based solution with the goal of *"a new multiprocessor architecture intended to make lock-free synchronization as efficient (and easy to use) as conventional techniques based on mutual exclusion."* [19].

After two decades of thorough TM research, it finally made to commercial hardware under the name *Hardware Transactional Memory* (HTM). All HTM systems provide the following machine instructions: begin, end and abort transactions.

- *Begin* instruction is used by the programmer to inform the HTM that a transaction has begun and all following reads and writes must be executed with atomicity and isolation guarantees.
- *End* instruction is called by the transaction to inform the TM that it is ready to commit.
- *Abort* instruction is used to abort a running transaction and call the abort handler, which is also activated upon a hardware triggered abort.

HTM detects conflicts with the granularity of a cache line, this differs from one processor to another. Table 2 shows the values for different processors that support HTM.

Although there exist different implementations of HTM, Nakaike et al. [27] show that no HTM outperforms all other for all workloads.

Table 2: HTM implementations of zEC12, Intel Core i7-4770 and POWER8. Adapted from [27]

Processor Type	zEC12	Intel Core i7-4770	POWER8
Conflict-detection granularity	256 bytes	64 bytes	128 bytes
Transactional-Load Capacity	1 MB	4 MB	8 KB
Transactional-Store Capacity	8 KB	22 KB	8 KB

zEC12 *IBMs zEnterprise EC12* (zEC12) [1] was the first commercial server to implement HTM. zEC12 uses L1 cache for conflict detection [21]. It provides *constrained transactions*, which are transactions guaranteed to eventually commit, avoiding the need of abort handlers. This characteristic allows zEC12 to perform well in highly contended scenarios [21].

POWER8 P8, also developed by *IBM* [23] uses *Content Addressable Memory* (CAM), a special type of memory, which keeps track of the address of cache lines accessed from within a transaction. CAM records all reads and writes, allowing for a quick search of all transactions using the searched word. Another

characteristic of P8 is its *suspend and resume transactions*. These allow the programmer a higher layer of liberty compared to other HTM since it allows the system to *suspend* the transaction. During suspend no data accesses are recorded by the HTM allowing the user to access clocks and counters outside its isolation ie. updating values visible to other transactions. The main downside of P8 is its low capacity compared to other HTM, as shown in the table 2.

TSX Intels TSX [33] provides two programming interfaces: *Hardware Lock Elision* (HLE) and *Restricted Transactional Memory* (RTM).

RTM is a simple HTM interface which allows programmers to specify a fall-back code if the HTM cannot successfully execute.

HLE Hardware Lock Elision (HLE) is an interface that implements *Speculative Lock Elision* (SLE). Basically, it provides the ability of transparently replacing the legacy lock acquire and release instructions with *XACQUIRE* and *XRELEASE* instructions. This transforms critical sections protected by locks into transactions that are executed speculatively. HLE is backward compatible, i.e., code developed with HLE will work on hardware without TSX support by falling back to pessimistic execution. HLEs drawback is its incapability of setting a custom fall-back code, using the original locks in case of failure.

Another drawback both Intel interfaces suffer of is the possibility of *spurious-aborts* due to data-conflict caused by pre-fetching cache-lines [27]. Although Intels pre-fetching feature can be disabled, doing so can degrade performance of other applications.

2.4 Lock Elision

Speculative Lock Elision(SLE), proposed by Rajwar et al. [28] is a novel technique which intends to dynamically spot and remove unnecessary serialization through locks, allowing previously locked critical sections to run concurrently. The concept of this paper is that frequent serialization lowers the performance of multi-threaded application, even if fine tuned. The main idea is that Hardware will dynamically identify synchronization operations, namely locks, and elide them, that is, instead of acquiring the lock the critical operation is executed as is. In the situation that two critical sections develop a conflict, the algorithm will fall-back to acquiring the lock pessimistically.

Legacy Code Ruan et al. in their paper [30] make use of *Lock Elision* (LE) as a way to allow legacy programs, previously implemented with lock-based synchronization, to elide the locks and implement the corresponding critical section as a transaction. They tested this implementation by changing the Compilers *cache_ lock and stats_ lock* instructions for atomic operations in TM. This allows legacy programs with limited performance in concurrency to be able to implement TM without the programmers having to consider the new complexity of perform changes to their code.

2.5 Hybrid Transactional Memory

Given the restrictions of existing HTM implementations, researchers have investigated an alternative approach, which goes under the name of *Hybrid Transactional Memory* (HyTM). In HyTM systems, transactions are first executed using HTM, yet fall-back to a STM if necessary, in an attempt to make the best use of both implementations. Unfortunately the simultaneous execution of HTM and STM induce high overheads to assure their correct synchronization [13].

HyNOrec *Hybrid No Ownership records* (HyNOrec), developed by Luke Dalessandro et al. [8], was created with the purpose of supporting concurrent hardware and software transactions while avoiding heavy instrumentation in hardware transactions. It uses *lazy subscription* and *eager conflict detection*.

As its name suggests, HyNOrec uses NOrec as its STM fall-back [9] which only requires a global clock, called TML. This allows for both HTM and STM to operate concurrently since both TM access and update this clock when writing.

Hardware Write transactions begin by reading TML to ensure they are subscribed to STM commit notifications, and increment it upon commit to signal software transactions. To avoid hardware-hardware conflicts due to TML changes, each processor core has its own local counter which each hardware transaction locally increments. This ensures a consistent snapshot, however it requires STM to increase its overhead as it must check the TML and each counter to guarantee its consistency with the HTM.

Invyswell Proposed by Irina Calciu et al. [4], *Invyswell* Invyswell is a HyTM that relies on a modified Inval-STM as the fall-back path of HTM. *Invyswell* uses *lazy subscription* and *commit-time invalidation*.

Inval-STM uses a novel method of validation called *commit-time invalidation* an *optimistic conflict detection* where each transaction stores its read and write-sets. During commit, the writer invalidates all conflicting transactions, giving itself priority. After finishing its *validation* it commits its changes to memory.

This simplifies the validation of other transactions, as they are immediately invalidated as if using *pessimistic conflict detection* without the regular conflict verification associated to this method.

To ensure its guarantees and increase the set of workloads where *Invyswell* performs well, five types of transaction were developed: *lightweight Hardware* (LiteHW), *bloom filter-based Hardware* (BFHW), *irrevocable Software* (IrrevSW), *speculative Software* (SpecSW) and *single global lock Software* (SglSW).

- LiteHW is a simple hardware transaction with no read or write software instrumentation. This allows for a faster execution. This benefit of LiteHW is also its downside as it is incapable of executing concurrently with software transactions.

– BFHW records its reads and writes, storing their memory location in Bloom filters. When finished, BFHW checks if the commit lock is free. If so, it increments the *hardware post commit lock* and commits. This lock prevents SpecSW from performing operations until it is free, allowing the BFHW to perform *commit-time invalidation*, with its recorded reads and writes, successfully.

– SpecSW is identical to *Inval-STM*. As with BFHW, SpecSW keeps track of accessed memory locations, both reads and writes, through Bloom filters. At commit time SpecSW performs *commit-time invalidation* with other SpecSW. Its main difference from *Inval-STM* is that it commits changes to memory before invalidating conflicting transactions.

– SglSW is a final transaction type used for small transactions the HTM does not support. Due to its small overhead SglSW is fast but does not allow for concurrent software executions as it acquires the SGL. It can however run in concurrency with HTM if it commits before BFHW and LiteHW check the SGL, as the HTM strong isolation detects and aborts if a data conflict occurs.

– IrrevSW is implemented for transactions that repeatedly could not commit in BFHW. As with SglSW it acquires the lock on start. All of its operations are immediately written to memory. During the execution of an IrrevSW, SpecSWs are disallowed to commit and BFHWs must check if they are conflicting and abort if needed.

Invyswell first tries transactions using HTM, running either in LiteHW or BFHW depending on other active transactions and the expected size of the transaction. If a transaction is not supported in HTM, it is immediately executed in SglSW. If the number of attempts a hardware transaction tries exceeds the defined retry policy, the transaction is tried in SpecSW. Finally if SpecSW continues to abort it is escalated to IrrevSW.

PhaseTM and Split Hardware Although not HyTM, *Phased Transactional Memory* (PhTM) [3] and *Split Hardware*(SplitTM) [24] use both HTM and STM. PhTM focuses on supporting several *phases* of the system, in which different TM-based synchronization schemes are used. It was presented with the following modes: Hardware, Software, Hybrid, Sequential and Sequential-NoAbort. This allows for adapting the employed TM implementation to the characteristics of the current workload. However, phase transitions take a stop the world approach: all threads must complete executing using the current synchronization mechanism, before they are allowed to start the new phase and use a different synchronization scheme.

SplitTM uses both STM and HTM by splitting an STM into multiple HTM segments, overcoming current HTM nesting issues. SplitTMs HTM sub-transactions

write to a thread-local log allowing the HTM to commit at any point of the parent transaction while ensuring isolation. HTMs also log their reads, allowing the parent transaction to maintain consistency as it can detect conflicts after the hardware transactions, where the reads occurred, have committed. Finally, on commit the parent transaction runs a hardware sub-transaction which writes all changes from the local write log to the main memory. Although allowing bigger transactions to be implemented in HTM, this implementations comes at the cost of instrumenting HTM transactions, tracking both reads and writes each HTM performs.

2.6 Self Tuning

As seen through the previous topics, TMs can be implemented in a variety of ways, each with their own set of parameters. These parameters are generally tuned manually, a time consuming and error prone task. Furthermore, it is not possible to implement a perfectly optimal configuration through a static manual tuning as workloads can vary over time. This motivated the investigation of self-tuning techniques for TM, of which I overview the following.

TinySTM When proposing *TinySTM* [14] Felber et al. noticed that some parameters of their algorithm, such as *hierarchical locking*, had to be fine tuned to each workload. In order to allow their algorithm to perform well in a larger set of workloads, they developed a *hill-climbing* tuning algorithm. Starting with a certain number of locks, the tuner periodically adapted these parameters attempting to acquire a more optimal value. This tuning algorithm proved capable of autonomously reaching throughput values close to those obtained by the team through static testing, optimized to the workload.

TSX Tuning Diegues and Romano [12] tackled the problem of automatically identifying the optimal number of times a transaction should be attempted in hardware, and how to react to capacity aborts, by activating the fall-back immediately or treating it as a conflict induced abort. The two sub-problems are tackled using different self-tuning algorithms, namely *hill-climbing* (with probabilistic jumps to avoid being trapped in local minimums) and *Upper Confidence Bound* (UCB) [5], a reinforcement learning algorithm that seeks an optimal trade-off between exploration of new configuration and exploitation of available knowledge. Its results showed that, as in *TinySTM*, self-tuning can reach results very close to those obtained through extensive off-line testing.

Green-CM Proposed by Shady et al. *Green-CM* [20] focuses on a *Contention Manager* directed mostly to optimize energy consumption, that is, avoiding aborts and implementing low consumption sleeps so as to reduce the energy

consumption of the TM. For this *Green-CM* proposes an energy efficient alternative for longer waits when blocked by a conflicting transaction. It separates waiting transactions into two types, *long waits* where they apply a time-based sleep, lowering consumption but also wait accuracy, and *short waits* were the algorithm applies a spin-based wait, a high energy consumption wait with high accuracy. To decide which *back-off policy* it should use, *Green-CM* makes use of both UCB and *hill-climbing*. Like TSX Tuning, *Green-CM* makes use of *hill-climbing* to explore the parameters searching for optimal configurations to the current workload. A problem of this method is that *hill climbing* continues to search for a better value even after arriving at the optimal configuration. In order to avoid changing to a less ideal configuration in subsequent oscillations, *Green-CM* uses a variant named *stabilizing* which functions as an UCB for the algorithm to avoid oscillating unnecessarily.

Proteus TM Didona et al. proposed *Proteus TM* [11], a self-tuning algorithm that focuses on adapting multiple parameters for optimal configurations. *Proteus TM* makes use of *Collaborative Filtering* (CF), a prominent technique in *Recommender Systems*, which attempts to obtain the best value for a user-defined *Key Performance Indicator* (KPI) and *Bayesian optimization* to profile the current workload to use CF with. KPI infers the ideal configuration of new workloads based on previously discovered optimal configurations for other workloads, as such, the algorithm is first implemented with an off-line profile of optimal configurations for a set of workloads. It then builds a matrix with the parameters to optimize in order to apply CF. Finally whenever a new workload appears, *Proteus TM* first attempts to profile the workload based on stored optimal configurations, using *Bayesian optimization*, and recommending the resulting KPI maxed configuration for the workload.

2.7 Read Write Lock Implementations

First described by Courtois et al. [7], the *Read/Writer Lock* (RWL) abstraction allows multiple readers to access the same value simultaneously, but locking the object from both readers and writers when a writer requests access to the value. Classic implementations of the RWL abstraction rely internally on mutex locks and semaphores.

The basic algorithm consists of two semaphores, one for active readers and one for writers. The reader, upon start, increments a waiting list to inform it is currently waiting to activate. It then verifies there are no writers active by checking the writer lock. If a writer is active, the reader will wait until the writer finishes. The reader then increments the semaphore and removes itself from the waiting list. After performing the critical section, the reader removes itself from the semaphore allowing writers to run again.

The writer begins by also publishing itself in a waiting list to inform it is ready to begin. It then verifies no reader is active and, if so, attempts to acquire the

writer lock. If successful it removes itself from the waiting list. Upon conclusion it removes itself from the writing lock, first signaling readers they may begin and afterwards writers.

The overall concurrent accesses allowed by this typical RWL can be seen in 3. In order to ensure a thread-safe access to the waiting lists a mutex lock is used.

The key challenge of RWLs design is how to minimize the additional overheads incurred with respect to plain mutex locks, while ensuring fair access to the lock to both readers and writers.

The main drawback of RWL is their poor scalability as they only have concurrency in reader-reader interactions as shown in table 3.

Table 3: Concurrent accesses allowed by typical RWL

	Reader	Writer
Reader	Yes	No
Writer	No	No

Big Reader Lock *Big Reader Lock* (BRLock)s [6] objective is to allow read-only transactions to function as fast as possible by locking a CPU-local spinlock. This implies a array of locks is created, one for each CPU. This algorithm was developed for read intensive workloads, as its objective is to increase reader throughput, resulting however in the reduction of writers throughput. The loss of writer throughput is due to the need of writers acquiring the full lock array to function.

PRWL *Passive Reader-Writer Locks* (PRWL) developed by Liu et al. [25], focuses in several points:

1. Readers do not need to share data between them, as such there is no shared state or the need of memory barriers if no writer exists.
2. In typical RWL writer use memory barriers to ensure version updates are visible to all readers/writers. To solve this situation without costly memory barriers PRWL uses *Inter-Processor Interrupts* (IPI) a special type of interrupt where one processor interrupts another, to force staggered readers to check the snapshot update.

HRWLE Proposed by Felber et al. [15], HRWLE is an algorithm, optimized for heavy-read workloads, which makes use of HTM concurrency capability to allow multiple writers to work concurrently via *hardware speculation*, enabling a different approach to the typical RWL system which only allows readers to run

concurrently. For this HRWLE makes use of P8s previously presented characteristic, *suspend and resume transactions*.

HRWLE works by treating readers and writers in distinct ways: writers are executed in HTM, allowing the system to automatically track conflicts between them. Conversely readers execute without any hardware instrumentation, hence avoiding the capacity limitations, which writers, by running in HTM, are subject to. Analogously to *BRLock*, in HRWLE readers announce their presence by flagging their presence in a thread-local variable (and ensuring the visibility of this update via a memory barrier).
To ensure correctness, no readers can be active during a writer commit. HTM however is known for its strong isolation, forcing the abortion in case of any data conflicts, such as flag verifications. Because of this isolation, in order to allow writers to commit more easily the algorithm makes use of P8s *suspend and resume* feature to *suspend* their transaction right before commit.

The writer can then access each readers flag to wait for each active reader to commit without aborting, maintaining a consistent snapshot and avoiding the writers abort due to flag value changes. After confirming each previously active reader has finished, the writer commits.

This allows for fairness between readers and writers as previously active transactions are always given priority to commit. If a reader activates itself during this *suspend* the correctness is still ensured, since any data reads conflicting with the writer will still provoke an abort of the writer, while the flag access itself does not.

As mentioned before, previously active readers are given priority in order to avoid starvation. Unfortunately, due to the HTMs strong isolation, writers cannot be given priority as any readers that access their data force an abort of the conflicting writer. HRWLE presents two methods of avoiding writer starvation: *Non-Speculative Transactions* and *Rollback-Only Transactions* (ROT).

If a writer has not successfully committed after a defined number of attempts, the algorithm tries to run it in ROT, a special HTM with minimized overhead. In this type of transaction the writer acquires SGL in *ROT-lock*, allowing only readers to execute concurrently. It also disables the *suspend* mechanism reducing the capacity needed and increasing the speed at which the transaction performs.

Non-Speculative Transactions is used by HRWLE when a transaction exceeds HTMs capacity or exceeds its maximum amount of tries defined in the configuration. In this case the writer acquires the SGL, waits for previous transactions to finish, performs its critical section and frees the lock afterwards. As the name implies, no other transaction may run during its execution.

Overall, HRWLE excels in workloads with high-capacity compared to the base HTM due to its fall-back paths. It also performs very well in high-contention workloads, compared to other RWL and HTM, as readers are un-instrumented and, in case of fall-back, ROT forces writer serialization, reducing SGL contention.

3 Work Proposal

As discussed, the pessimistic nature of lock-based synchronization mechanisms can unnecessarily limit parallelism and prevent tapping the full potential of modern massively parallel architectures.

By exploiting P8s *suspend and resume* supports, HRWLE can achieve excellent performance in many types of workloads. Its use of *suspend and resume* in writers allows it to verify flags otherwise bound to force an abort in typical HTM, enabling a higher level of concurrency and fairness between HTM and readers. It also allows for the un-instrumentation of readers, as HTM conflict detection and the reader flags ensure a consistent snapshot is always maintained. Unfortunately, due to its reliance on *suspend and resume* mechanism, HRWLE is limited to P8 processors, which impedes its usability in other HTM-equipped processors, in particular Intel CPUs.

This work proposes the development of a HTM supporting un-instrumented readers, much like HRWLE, but making only use of "standard" HTM instructions, that is *BEGIN*, *END* and *ABORT*, so as to make it capable of running in any HTM. This algorithm will be implemented for Intel TSX.

3.1 Initial Solution

During the development of this report an initial version of the solution has been developed. Algorithm 1 shows the pseudo-code for this initial version. It was developed as a proof of concept in order to confirm that it is feasible to run un-instrumented readers along with HTM writers while still ensuring correctness and achieving performance gains. Due to the best effort nature of current HTM implementations, a *single_global_lock* (SGL) is used as a fall-back in case a transaction cannot complete successfully in HTM.

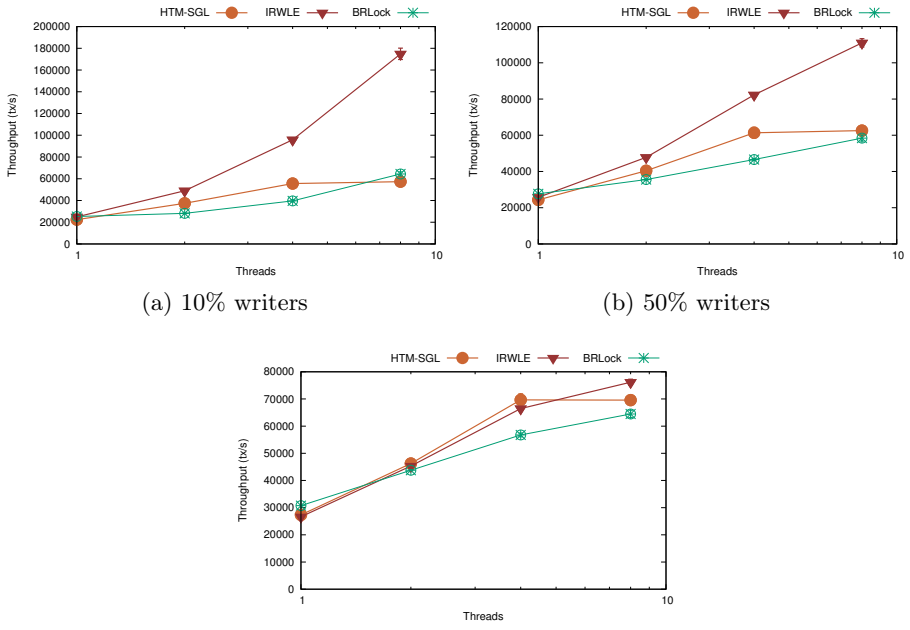
The algorithm begins by separating readers and writers on acquire call, similar to read-write lock interface.

Upon read acquire, the algorithm executes outside the HTM. However if the call originates from a writer, the algorithm directs it to the HTM synchronization method as usual. When a reader starts (line 6), it first checks that the SGL is free. If so, it publicizes itself by updating its corresponding *isReader* counter. It then proceeds to run its the critical section and, on finish (line 15), concludes its operation by disabling its *isReader* counter, allowing other transactions to know it is no longer actively reading. If the *single_global_lock* is locked, the reader must then wait for its unlock to proceed as above. A writer however must proceed in a different way (line 17), it first confirms that the SGL is free, as with reader, starts in a HTM transaction and then re-reads the SGL. This serves to add the SGL to its read set, ensuring the writer aborts if the lock is acquired by some other thread during our writers transaction. After running its critical section and ready to commit (line 33), it first checks all *isReader* counters in order to confirm no reader is active (line 36). Only if no reader is found active is the writer allowed to commit. If a reader is active, the writer must forcibly be aborted as it is isolated and cannot instruct the reader to stop.

For this implementation to function correctly, while ensuring atomicity, the algorithm makes use of the fact that HTM detects conflicts between both concurrent transactions and non-transactional read/write accesses. This ensures that all writers can be executed concurrently as HTM detects their collisions. Readers however, are not instrumented and do not run in HTM. As such, they are prone to reading inconsistent states if a writer commits while a reader is active. This, however, is avoided by forcing writers to commit only when there are no active readers. This is achieved by readers publicizing their existence, through the shared flag *isReader* (line 2).

If a reader is active when the writer is ready to commit, the writer must abort as it cannot know which data the reader transaction has seen and may see, making it possible to generate a inconsistent snapshot for the reader. It is worth noting here that if a reader starts after the writer checks for active readers but before committing, the writer will abort due to HTMs conflict detection since the reader begins its operation by updating its *isReader* counter.

3.2 Experimental Results

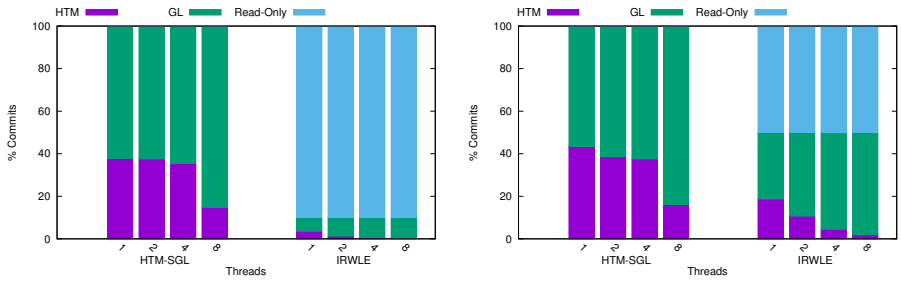


(a) 10% writers

(b) 50% writers

(c) 90% writers

Fig. 1: Hashmap testing with high Capacity and low Contention for 10%(a) 50%(b) and 90%(c) writers



(a) 10% writers (b) 50% writers

Fig. 2: Comparison of commit percentages between suspend and IRWLE in a Hashmap with high capacity low contention at 10%(a) and 50%(b) writers

To validate this work proposal, a preliminary performance evaluation was performed in order to compare between our initial algorithm, noted as Intel-RWLE (IRWLE), and some of the algorithms described above, more specifically the Big Readers Lock (BRLock) and a solution purely based on HTM (noted as HTM-SGL). The considered workload generates update transactions with probability 10% 1a, 50% 1b and 90% 1c.

Update transactions operate on a hash map that has 1000 buckets, each containing 800 items. More precisely, the input value is hashed to a bucket and the bucket is scanned to search for that value. If the value is found, the value is updated, else the value is added to the bucket. The readers, on the other hand, simply scan the hash map in search of the input value and return it if they find it. This workloads parameters were set to generate very limited contention and high probability of incurring a capacity exception.

The results obtained, figure 1, proves our proof of concept previously explained, revealing that our algorithm can achieve remarkable performance gains (2x to 3x) with respect to both HTM-SGL and BRLock for both the case of 10% and 50% of update transactions.

Since the reader, in our initial algorithm, is not instrumented and is not run in *transactional memory*, its throughput is higher due to its almost inexistent overhead, compared to typical HTM transactions and locks as seen in 1a and 1b.

We can further compare our performance with HTM-SGL by observing the commit percentage in figure 2. With the increase of concurrent threads, both algorithms see a decrease of HTM usage and an increase in SGL commits, our algorithm however only sees this occur in writers since readers are un-instrumented.

Un-instrumented readers can always commit with no conflicts, compared to writers that may abort, performing always at the ideal throughput. Overall our algorithm acquires much less frequently SGL compared to HTM-SGL, allowing our system to perform concurrently longer, thus our increase in throughput.

3.3 Improvements

As seen in our algorithm 1, there are several aspects to correct and improve. The first issue to review is its fairness. As most workloads are reader focused, the algorithm was developed with focus on readers. However, in its current form the more concurrent readers are running the more likely it is for a writer to be unable of successfully committing since readers cannot be abort and can begin at any moment. This forces writers to acquire the fall-back lock disabling concurrency in the system, as seen in the results 2, where readers have exactly the percentage of commits expected but writers have a much lower use of HTM compared to HTM-SGL since readers continuously abort them.

In order to solve this issue, we plan to introduce a scheduling technique based on the idea of estimating execution times of readers and writers and exploit this information to mitigate contention and ensure fairness. By publicizing end times of readers and writers, readers can predict conflicts with writers and postpone their start, in order to wait for the writers to successfully commit before beginning. Writers can also check the expected end-times of read critical sections and activate their hardware transaction only if it is expected to end after the commit of all readers that are found already active.

This solution however generates another issue, how to predict in a precise manner the end-time of both readers and writers. Our proposed solution is to develop an auto-tuning algorithm which reads the *Time Stamp Counter* (TSC) of cores and dynamically tunes its prediction. We must also verify if reading TSC influences or not the performance of the proposed algorithm.

3.4 Benchmarks

To evaluate our algorithm we shall use several benchmarks, we describe the following:

Synthethic the first benchmark I plan to use is a synthetic benchmark that mimics the behavior of a concurrent hash-map, analogously to the one used for producing the results in figures 1 and 2. By tuning the number of buckets and entries in the them, this benchmark allows for precise control of contention and capacity abort probabilities, which are expected to have a major impact on the performance of the proposed solution.

STMBench7 Proposed by Guerraoui et al. [18], *STMBench7* simulates complex applications, such as CAD, composed of several different and large data structures, such as indexes and graphs. It is one of the most complex TM benchmarks, allowing flexible customization of the generated workload and extensively testing the proposed algorithms in heterogeneous settings.

TPC-C *TPC-C* benchmark is used to simulate a complete online transaction environment and is representative of a warehouse supplier application. *TPC-C* uses five different types of transactions (new-order, payment..), with very diverse profiles, such as long read-only transactions, long and contention-prone vs short and almost contention-free update transactions. Also this benchmark can be flexibly configured to generate heterogeneous workloads.

Kyoto Cabinet Finally, I am going to consider *Kyoto Cabine* [22], a commercial C++ database management library. In particular, I will focus on the in-memory variant *KyotoCacheDB*. Internally, it breaks the database into slots, where each slot is composed of buckets and each bucket is a search tree. To synchronize database operations, *KyotoCacheDB* uses a single global read-write lock. As such, this is an ideal use case for lock elision techniques specialized to deal with read-write locks.

3.5 Work Plan

In this section is the work plan to be followed during my dissertation. I propose the following deadlines:

- 20/10/2017 : Integration of mechanism to ensure fairness between readers and writers
- 24/11/2017 : Development of scheduling policy based on transaction duration estimation
- 9/02/2017 : Design and evaluation of different transaction prediction policies
- 9/03/2017 : Evaluate with micro and macro benchmarks the solution both in Intel and P8, considering single-socket and multi-socket deployment scenarios
- 13/04/2018 : Completion of a report describing the results of this dissertation
- 27/04/2018 : Completion of the dissertation

4 Conclusion

The report explains the current state-of-art of *Transactional Memory*, detailing its features and most common hardware and software implementations. It also explains different implementations of read-write locks and lock elision. More specifically, *Hardware Read Write Lock Elision* which makes use of the *suspend and resume* feature of P8 processor and allows for supporting un-instrumented readers. It also details on Self-Tuning as a method of optimizing *Synchronization Methods* according to the nature of workloads. Next, it proposes the implementation of a *Hardware Lock Elision* technique supporting un-instrumented readers while retaining compatibility with Intels HTM implementation, i.e., not requiring the *suspend and resume* feature. It also presents an initial version of the algorithm which has several aspects to improve but proves the practicality of this concept. An improved implementation will be developed by developing an

auto-tuning algorithm to enhance the fairness between readers and writers. The expectation is to obtain experimental results proving both practicality and performance in comparison to current *Transactional Memory* implementations and state of the art read-write lock implementations.

References

1. z/Architecture Principles of Operation. SA22-7832-09.
2. Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking concurrency. *Queue*, 4(10):24–33, December 2006.
3. Moir Bussam, Lev. Second acm sigplan workshop on transactional computing (transact 07). 2007.
4. Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: A hybrid transactional memory for haswell's restricted transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 187–200, New York, NY, USA, 2014. ACM.
5. Alexandra Carpentier, Alessandro Lazaric, Mohammad Ghavamzadeh, Rémi Munos, and Peter Auer. Upper-confidence-bound algorithms for active learning in multi-armed bandits. In *Proceedings of the 22Nd International Conference on Algorithmic Learning Theory*, ALT'11, pages 189–203, Berlin, Heidelberg, 2011. Springer-Verlag.
6. Jonathan Corbet. Big reader locks. https://lwn.net/Articles/378911.
7. P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Commun. ACM*, 14(10):667–668, October 1971.
8. Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. *SIGPLAN Not.*, 46(3):39–52, March 2011.
9. Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: Streamlining stm by abolishing ownership records. *SIGPLAN Not.*, 45(5):67–78, January 2010.
10. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.
11. Diego Didona, Nuno Diegues, Anne-Marie Kermarrec, Rachid Guerraoui, Ricardo Neves, and Paolo Romano. Proteustm: Abstraction meets performance in transactional memory. *SIGOPS Oper. Syst. Rev.*, 50(2):757–771, March 2016.
12. Nuno Diegues and Paolo Romano. Self-tuning intel restricted transactional memory. *Parallel Comput.*, 50(C):25–52, December 2015.
13. Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 3–14, New York, NY, USA, 2014. ACM.
14. Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.
15. Pascal Felber, Shady Issa, Alexander Matveev, and Paolo Romano. Hardware read-write lock elision. In *Proceedings of the Eleventh European Conference on*

Computer Systems, EuroSys '16, pages 34:1–34:15, New York, NY, USA, 2016. ACM.

16. C. Ferri, S. Wood, T. Moreshet, R. Iris Bahar, and M. Herlihy. Embedded-tm: Energy and complexity-effective hardware transactional memory for embedded multicore systems. *Journal of Parallel and Distributed Computing*, 70(10):1042 – 1052, 2010. Transactional Memory.
17. E. Gaona, R. Titos, J. FernÃąndez, and M. E. Acacio. On the design of energy-efficient hardware transactional memoryâĂĽsystems. *Concurrency and Computation: Practice and Experience*, 25(6):862–880, 2013.
18. Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: A benchmark for software transactional memory. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM.
19. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.
20. Shady Issa, Paolo Romano, and Mats Brorsson. Green-cm: Energy efficient contention management for transactional memory. In *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP)*, ICPP '15, pages 550–559, Washington, DC, USA, 2015. IEEE Computer Society.
21. Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for ibm system z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 25–36, Washington, DC, USA, 2012. IEEE Computer Society.
22. FAL Labs. Kyoto cabinet: A straightforward implementation of DBM, 2011. http://fallabs.com/kyotocabinet/.
23. H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the ibm power8 processor. *IBM Journal of Research and Development*, 59(1):8:1–8:14, Jan 2015.
24. Yossi Lev and Jan-Willem Maessen. Split hardware transactions: True nesting of transactions using best-effort hardware transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 197–206, New York, NY, USA, 2008. ACM.
25. Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 219–230, Philadelphia, PA, 2014. USENIX Association.
26. A. Mericas, N. Peleg, L. Pesantez, S. B. Purushotham, P. Oehler, C. A. Anderson, B. A. King-Smith, M. Anand, J. A. Arnold, B. Rogers, L. Maurice, and K. Vu. Ibm power8 performance features and evaluation. *IBM Journal of Research and Development*, 59(1):6:1–6:10, Jan 2015.
27. Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. *SIGARCH Comput. Archit. News*, 43(3):144–157, June 2015.
28. Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

29. Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 284–298, Berlin, Heidelberg, 2006. Springer-Verlag.
30. Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing legacy code: An experience report using gcc and memcached. *SIGPLAN Not.*, 49(4):399–412, February 2014.
31. Martin Schindewolf, Barna Bihari, John Gyllenhaal, Martin Schulz, Amy Wang, and Wolfgang Karl. What scientific applications can benefit from hardware transactional memory? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 90:1–90:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
32. Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raúl Silvera, and Maged M. Michael. Evaluation of blue gene/q hardware support for transactional memories. *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 127–136, 2012.
33. Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM.

Algorithm 1 initial version of proposed algorithm

```
1: Shared variables:
2:     isReader[N] ← {0, 0, ..., 0}        ▷ Boolean counter for each thread to identify itself as reader or writer
3:     single_global_lock

4: Local variables:
5:     tid ∈ [0..N]                        ▷ Identifier of current thread

6: function READ_LOCK
7:     while 1 do
8:         isReader[tid]= 1                ▷ Updates its Status to inform it is Active
9:         if IS_LOCKED(single_global_lock) then
10:            isReader[tid]= 0            ▷ Updates its Status to inform it is Inactive
11:            Wait
12:            Continue
13:        MemBarrier()                    ▷ Memory Barrier to ensure correct order of instructions
14:        Break

15: function READ_UNLOCK
16:    isReader[tid]= 0                    ▷ Updates its Status to inform it is Inactive

17: function WRITE_LOCK
18:    retries = 0
19:    while 1 do
20:        if IS_LOCKED(single_global_lock) then
21:            Wait
22:        status = TX_BEGIN
23:        if status == tx_started then    ▷ Confirm the GL is free
24:            if IS_LOCKED(single_global_lock) then
25:                TX_ABORT
26:        if status == tx_abort then      ▷ If aborted register the attempt and retry
27:            retries + +
28:            if retries == MAX_RETRIES then   ▷ Acquire Lock as last resort
29:                Lock( single_global_lock)
30:                for i ← 0 to N−1 do      ▷ Wait for all threads to finish
31:                    if isReader[i] ≠ 0 then
32:                        WAIT

33: function WRITE_UNLOCK
34:    SYNCHRONIZE                         ▷ Confirm no Readers are active
35:    TX_COMMIT                           ▷ Write back updates

36: function SYNCHRONIZE
37:    for i ← 0 to N−1 do                 ▷ Check for active Readers...
38:        if isReader[i] ≠ 0 then
39:            TX_ABORT
```