Speculative Read-Write Locks

Tiago João dos Santos Lopes

Mestrado em Engenharia Informática e Computadores

Information Systems and Computer Engineering

Supervisor: Dr. Paolo Romano

Draft

May, 2018

Resumo

Memória Transacional (TM) é uma abstração para programação paralela promissora, que tem sido recentemente implementada em hardware por produtores principais como a Intel e IBM. Hardware Transactional Memory (HTM) expõe aos programadores, através de uma extensão dedicada das suas instruções de processador, uma implementação assistida por hardware, e altamente eficiente, da abstração de transações atómicas.

Vários trabalhos recentes têm mostrado que HTM consegue reduzir significativamente o custo de sincronização sofrido por aplicações paralelas em várias áreas. Infelizmente, também já foi identificado por vários estudos que implementações HTM atuais sofre de graves limitações, em grande parte devido á natureza restrita de mecanismo *best-effort* que o hardware utiliza.

Estas restrições do design limitam a aplicabilidade de HTM de várias formas: não só restringe o número de posições de memória acessíveis pela transação, também faz com que as transações de hardware incapazes de de suportar eventos tais como chamadas de sistema, perdas de processador *context switching* e pedidos de interrupção. Desta forma as restrições mencionadas tornam os sistemas HTM atuais incapazes de servir como mecanismos de sincronização polivalentes, o que limita a sua utilização.

Este trabalho tenta combater exatamente este problema apresentando *Speculative Read Write Lock* (SpRWLock), um novo mecanismo de sincronização baseado em HTM com um ponto-chave: permite blocos atómicos de leitura executa fora do sistema de transações em hardware, effectivamente poupando-as das limitações

existentes nas atuais implementações HTM.

Speculative Read Write Lock (SpRWLock) combina duas técnicas novas com o objetivo de asseguras a *safety* e maximizar eficiência.

SpRWLock preserva a *safety* de blocos atómicos de leitura, correndo fora do alcance de transações de hardware, através de uma simples, mas supreendentemente eficiente, técnica: obrigas as transações de escrita, que executam em HTM, que verifiquem, no momento de submissão, verificar se existem blocos atómicos de leitura ativos, e permitindo a submissão dos escritores apenas se nenhum for encontrado; caso contrário abortam a transação HTM.

Esta técnica consegue obter impressionantes (até 6×) ganhos de processamento comparado ao sistema base HTM em cargas de trabalho (*workload*) que contenham grandes blocos atómicos de leitores. No entanto esses ganhos de processamento são obtido á custa de um aumento no periodo de latência de blocos de escritura atómicos, que chegam a abortar diversas ver (e teóricamente ser privados de executar) em *workloads* onde leitores grandes são predominantes.

SpRWLock foca neste problemas complementando o algoritmo base acima descrito com dois esquemas ad-hod de calendarização, as quais referimos de sincronização de leitores e sincronização de escritores.

Avaliamos SpRWLock através de um estudo experimental extensivo utilizando as implementações HTM disponíveis nos CPUs *Broadwell* da Intel e *Power8* da IBM e abrangendo micro-benchmarks destinadas a avaliar a sensibilidade da solução proposta a um spectro de *workloads*, bem como benchmarks padrão (TPC-C, STMBench7) e aplicações reais (KyotoDB). Os resultados do nosso estudo mostram que SpRWLock consegue ganhos de desempenho até 15× em relação tanto a soluções HTM base, como também ao estado de arte atual, Implementações de trincos em leitores-escritores de forma não especulativa *non-speculative read-write lock implementations*.

Palavras-Chave

memória transacional, sincronização, leitores-escritores, elisão de trincos, hardware

Abstract

Transactional Memory (TM) is a promising abstraction for parallel programming, which has recently been implemented in hardware by mainstream like Intel and IBM. Hardware Transactional Memory (HTM) provides a highly-efficient, hardware-assisted implementation of the abstraction of atomic transaction, long used in the context of database systems, and now exposed to programmers/compilers via a dedicated extension of the processor instruction set.

A number of recent works have shown that HTM can reduce significantly the synchronization overheads incurred by parallel applications in various application domains. Unfortunately, though, several studies have also highlighted that existing HTM implementations suffer of severe limitations, stemming from the inherently restricted nature of the best-effort hardware mechanisms that they employ.

Such a design approach limits the applicability of HTM in a number of ways: not only it explicitly restricts the maximum amount of memory positions that can be accessed by transaction, but also makes hardware transactions unable to withstand events that lead to scratching the processor's cache, which includes, notably, system calls, context switches and interrupt requests (including periodic timer interrupts raised for OS scheduling purposes). Overall, these restrictions make current HTM systems unfit to serve as a general-purpose synchronization mechanism, significantly limiting the scope of their applicability.

This work aims at tackling precisely this issue by introducing *Speculative Read Write Lock* (SpRWLock), a novel HTM-based synchronization primitive that provides a key benefit: allowing read-only atomic blocks to execute outside the scope

of any hardware transaction, thus, effectively sparing them from the inherent limitations affecting existing HTM implementations.

SpRWLock combines two key novel techniques aimed, respectively, at ensuring safety and at maximizing efficiency.

SpRWLock preserves safety of read-only atomic blocks, which run outside the scope of hardware transactions, by using a simple, yet surprisingly effective, technique: it requires update transactions, which execute using HTM, to check, at commit time, whether there are any active read-only atomic blocks, and allows them to commit only if none is found; forcing them to abort otherwise.

This technique can yield remarkable (up to 6×) throughput gains over plain HTM in workloads that have long read-only atomic blocks. However, these throughput gains are achieved at the cost of an increased latency of update atomic blocks, which can suffer from frequent aborts (and theoretically from starvation) in read-dominated workloads.

SpRWLock addresses these shortcomings by complementing the above base algorithm with two ad-hoc scheduling schemes, which we refer to as reader synchronization and writer synchronization.

We evaluated SpRWLock via an extensive experimental study conducted using the HTM implementations available on Intel's Broadwell and IBM's Power8 CPUs and encompassing synthetic micro-benchmarks aimed at assessing the sensitivity of the proposed solution to a broad spectrum of workloads, as well as standard benchmarks (TPC-C, STMBench7) and real applications (KyotoDB). The results of our study shows SpRWLock can yield throughput gains of up 15× with respect to both plain HTM -based solutions, as well as state of the art, non-speculative read-write lock implementations.

Keywords

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

Contents

List of Figures

List of Algorithms

Glossary

BFHW *bloom filter-based hardware*. 1, 21, 22

BRLock *Big Reader Lock*. 1, 9, 48, 50, 51, 53, 55, 56

CAM *Content Addressable Memory*. 1, 16

CF *Collaborative Filtering*. 1, 25

CPU *Computer Processing Unit*. 1

HLE *Hardware Lock Elision*. 1, 17

HRWLE *Hardware Read-Write Lock Elision*. iii, vii, 1, 17–19, 27, 41, 48, 50, 51

HTM *Hardware Transactional Memory*. iii, vii, 1, 2, 15–20, 22, 23, 27, 60

HyNOrec *Hybrid No Ownership records*. 1, 20

HyTM *Hybrid Transactional Memory*. 1, 20, 22

IPI *Inter-Processor Interrupts*. 1, 9

IrrevSW *irrevocable software*. 1, 21, 22

KPI *Key Performance Indicator*. 1, 25

LE *Lock Elision*. 1

LiteHW *lightweight hardware*. 1, 21, 22

NOrec *No Ownership records*. 1, 14, 20

P8 *POWER8*. 1, 16–18

PhTM *Phased Transactional Memory*. 1, 22

PRWL *Passive Reader-Writer Locks*. 1, 9

RAM *Random Access Memory*. 1

ROT *Rollback-Only Transactions*. 1, 18, 19

RTM *Restricted Transactional Memory*. 1, 17

RWL *Read/Writer Lock*. 1, 8, 9, 17, 19, 41, 48, 50

SGL *Single Global Lock*. 1, 2, 18, 19, 22

SglSW *single global lock software*. 1, 21, 22

SLE *Speculative Lock Elision*. 1, 17, 19

SpecSW *speculative Software*. 1, 21, 22

SpRWLock *Speculative Read Write Lock*. iii, vii, xi, xiii, 1, 3–5, 27–31, 33–35, 41–56, 59, 60

STM *Software Transactional Memory*. 1, 2, 12–14, 20, 22, 23

TL2 *Transactional Locking II*. 1, 13

TLE Transactional Lock Elision. 1, 46, 47, 49, 50

TM *Transactional Memory*. iii, vii, 1, 2, 10–12, 15, 19, 20, 23, 24, 27, 60

TML *Transactional Mutex Lock*. 1, 14, 15, 20

TSC *Time Stamp Counter*. 1

TSX *Transactional Synchronization Extensions*. 1, 17

UCB *Upper Confidence Bound*. 1, 24, 25

zEC12 *IBMs zEnterprise EC12*. 1, 16

Chapter 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.

Transactional Memory (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 synchroniza-

tion, 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 (*Hardware Transactional Memory* (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 [1–6] 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 [4, 7–9], 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.

Such a design approach limits the applicability of HTM in a number of ways: not only it explicitly restricts the maximum amount of memory positions that can be accessed by transaction, but also makes hardware transactions unable to withstand events that lead to scratching the processor's cache, which includes, notably, system calls, context switches and interrupt requests (including periodic timer interrupts raised for OS scheduling purposes). Overall, these restrictions make current HTM systems unfit to serve as a general-purpose synchronization mechanism, significantly limiting the scope of their applicability.

This work aims at tackling precisely this issue by introducing *Speculative Read Write Lock* (SpRWLock), a novel HTM-based synchronization primitive that provides a key benefit: allowing read-only atomic blocks to execute outside the scope

of any hardware transaction, thus, effectively sparing them from the inherent limitations affecting existing HTM implementations. SpRWLocks name stems from the fact that it exposes to programmers the familiar interface of a classic read-write lock and can, therefore, be seen as a specialized HTM-based technique for eliding this type of locks in legacy applications. However, SpRWLock can also be straightforwardly employed in applications that assume a transactional API by mapping the beginning of a read-only or an update transaction to a request for acquiring a read or write lock, respectively.

SpRWLock combines two key novel techniques aimed, respectively, at ensuring safety and at maximizing efficiency.

SpRWLock preserves safety of read-only atomic blocks, which run outside the scope of hardware transactions, by using a simple, yet surprisingly effective, technique: it requires update transactions, which execute using HTM, to check, at commit time, whether there are any active read-only atomic blocks, and allows them to commit only if none is found; forcing them to abort otherwise. The correctness of this approach hinges on two key properties of HTM:

- HTM externalizes the memory writes produced by an update transaction only if it successfully commits, making them visible to both transactional and non-transactional code atomically. This ensures that read-only atomic blocks never observe uncommitted (e.g., intermediate) writes of a concurrent update transaction.

- HTM detects conflicts between transactional and non-transactional code in an eager fashion, triggering the immediate abort of the former — a property that is known as *strong isolation* in the literature [10]. This property avoids data races, which might otherwise occur if a read-only atomic block started, after its state was checked (and found inactive) by a concurrent update transaction. As illustrated in Fig. 1.1, if the update transaction was to be allowed to commit the read-only atomic block could observe an inconsistent snapshot, e.g., by returning different values upon two subsequent reads of the same memory

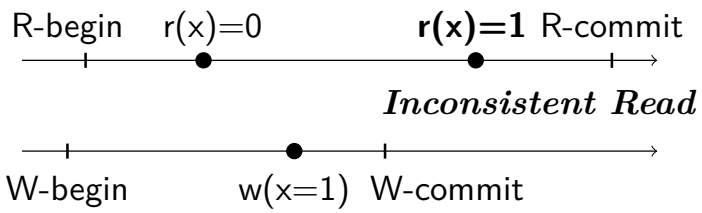

Figure 1.1: Allowing a writer to commit while there is an active reader may lead to inconsistent snapshots

position. SpRWLock prevents such a scenario by leveraging the strong isolation property of HTM, which ensures that if a read-atomic block alters its own state after it has been checked by a concurrent update transaction, then the latter will be immediately aborted.

As we will show, the technique described above can yield remarkable (up to 6×) throughput gains over plain HTM in workloads that have long read-only atomic blocks. However, these throughput gains are achieved at the cost of an increased latency of update atomic blocks, which can suffer from frequent aborts (and theoretically from starvation) in read-dominated workloads.

SpRWLock addresses these shortcomings by complementing the above base algorithm with two ad-hoc scheduling schemes, which we refer to as reader synchronization and writer synchronization.

The reader synchronization scheme requires that read-only atomic blocks, before starting, wait for the completion of active concurrent update atomic blocks, if any. This technique not only reduces the likelihood for an update transaction to abort due to the existence of a concurrent read atomic block; it also guarantees an important fairness property for write atomic blocks, i.e., it ensures that if a write atomic block is activated before a read atomic block, then the former cannot be aborted by the latter. The reader synchronization scheme of SpRWLock is further enhanced by allowing read-only atomic blocks to shortcut their initial waiting phase in case they find some other read-only atomic block already waiting: in such a case, the last

activated read-only atomic block joins the one already waiting and starts as one the latter does. This brings two advantages: reducing the average duration of the readers' waiting phase — and, hence, their latency — and striving to minimize the duration of time windows during which some reader is active by aligning their start time — which increases the chance for update atomic blocks to commit successfully.

The writer synchronization scheme aims, instead, at optimizing the scheduling decision on when to activate an update atomic block by pursuing a twofold goal: on one hand, postponing the activation of update atomic blocks in order to reduce the chances that they have to abort, eventually, due to a concurrent read atomic block; on the other hand, activating update atomic blocks as early as possible, to maximize concurrency already active read atomic blocks. This is achieved by estimating, at run-time, the average duration of atomic blocks, and accordingly delaying the activation of an update atomic block in order to maximize the chances that it requests to commit shortly after the last concurrent read atomic block completed.

We evaluated SpRWLock via an extensive experimental study conducted using the HTM implementations available on Intel's Broadwell [11] and IBM's Power8 [12] CPUs and encompassing synthetic micro-benchmarks aimed at assessing the sensitivity of the proposed solution to a broad spectrum of workloads, as well as standard benchmarks (TPC-C [13], STMBench7 [14]) and real applications (KyotoDB [15]). The results of our study shows SpRWLocks throughput can reach 15× typical TM systems in some of the standard benchmarks, at the cost of reader latency.

Chapter 2

Related Work

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 [16]. Furthermore, it compromises a key desirable property of software: composability [16]. Using Locks as a synchronization method not only disables concurrent access to such values, but also delays the threads themselves with its additional overhead during its normal workloads.

A 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. In order to achieve this, transactions have to be Atomic, Consistent, Isolated and Durable (ACID).

These features are also essential for parallel programming. Atomicity requires that changes done within a transaction appear as all or none to other code. Consistency demands that as data changes the database always remains in a valid state. Isolation ensures that all changes done from within a transaction must remain in-

visible to all other transactions. Durable as in case of failure the system can either recover entirely or discard the changes committed by the failing transactions.

2.1 Read Write Lock Implementations

First described by Courtois et al. [17], 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 2.1. 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 2.1.

Table 2.1: Concurrent accesses allowed by typical RWL

	Reader	Writer
Reader	Yes	No
Writer	No	No

2.1.1 Big Reader Lock

Big Reader Lock (BRLock)s [18] 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.

2.1.2 PRWL

Passive Reader-Writer Locks (PRWL) developed by Liu et al. [19], 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.

2.1.3 RCU

Read-Copy-Update (RCU) [20] is an alternative synchronization mechanism that targets read-dominated workloads. Unlike RWLs, with RCU, a read-only critical section does not need to acquire any mutex, it just flags itself, using a memory barrier, at the beginning and end of critical section. To ensure correctness, a writer

modifying shared data, would create a copy of the data and apply the modifications to the copy. Readers that existed prior to the write would continue to access the older, unmodified data, while new readers get to witness the updates. Only when all readers that existed before the writer have completed their critical sections, the unmodified data is replaced by the copy.

2.2 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) complying with ACID.

TM is a parallel programming paradigm that avoids the pitfalls of traditional locking techniques while promising the performance of fine-grained locking [16]. 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 eagerly checks the transactions accessed values. This allows for a 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 loss that *pessimistic conflict detection* has due to 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* the system locally stores read addresses and values, 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 and *invisible*, where readers do not inform other active transactions of which shared memories they read, forcing writers to check if reads and writes are complete and a consistent snapshot of the system is maintained on commit.

2.3 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 2.2 shows some popular and efficient STM implementations.

Table 2.2: Popular STM characteristics

	Data Versioning	Conflict Detection	Granularity
TL2 [21]	Lazy	Optimistic	Word/Object
TinySTM [22]	Lazy/Eager	Pessimistic	Word
NORec [23]	Lazy	Pessimistic	Value

2.3.1 Transactional Locking II

Transactional Locking II (TL2), proposed by Dave Dice et al. [21], 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 re-validates 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.

2.3.2 TinySTM

Pascal et al. later proposed *TinySTM* [22], a word-based variant of LSA [24]. *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.

2.3.3 NOrec

No Ownership records (NOrec) presented by Luke Dalessandro et al. [23] 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.

2.4 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."* [25].

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 2.3 shows the values for different processors that support HTM.

Table 2.3: HTM implementations of zEC12, Intel Core i7-4770 and POWER8. Adapted from [5]

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

Although there exist different implementations of HTM, Nakaike et al. [5] show that no HTM outperforms all other for all workloads.

2.4.1 zEC12

IBMs zEnterprise EC12 (zEC12) [7] was the first commercial server to implement HTM. zEC12 uses L1 cache for conflict detection [26]. 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 [26].

2.4.2 POWER8

POWER8 (P8), also developed by *IBM* [27] 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 2.3.

2.4.3 TSX

Intels *Transactional Synchronization Extensions* (TSX) [28] 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 [5]. Although Intels pre-fetching feature can be disabled, doing so can degrade performance of other applications.

2.4.4 HRWLE

Proposed by Felber et al. [29], *Hardware Read-Write Lock Elision* (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.

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* mode, allowing only writers to execute concurrently. As it uses a minimized overhead, ROT transactions perform faster then HTM.

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.

2.5 Lock Elision

Speculative Lock Elision(SLE), proposed by Rajwar et al. [30] 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.

2.5.1 Legacy Code

Ruan et al. in their paper [31] make use of SLE 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 run in HTM without the programmers having to consider the new complexity of perform changes to their code.

2.6 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 [1].

2.6.1 HyNOrec

Hybrid No Ownership records (HyNOrec), developed by Luke Dalessandro et al. [32], 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 [23] 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.

2.6.2 Invyswell

Proposed by Irina Calciu et al. [33], *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 be-

fore 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.

2.6.3 PhaseTM and Split Hardware

Although not HyTM, *Phased Transactional Memory* (PhTM) [34] and *Split Hardware*(SplitTM) [35] 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.

2.7 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.

2.7.1 TinySTM

When proposing *TinySTM* [22] 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.

2.7.2 TSX Tuning

Diegues and Romano [8] 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) [36], 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.

2.7.3 Green-CM

Proposed by Shady et al. *Green-CM* [37] 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.

2.7.4 Proteus TM

Didona et al. proposed *Proteus TM* [38], 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.

Chapter 3

Algorithm

This chapter presents *Speculative Read Write Lock* (SpRWLock), a novel HTM-based techniques for eliding read-write lock, which, analogously to HRWLE, supports the concurrent execution of un-instrumented readers âĂŤ hence sparing them from HTMâĂŹs capacity limitations âĂŤ as well of writers using HTM. The key novelty of SpRWLock is that, unlike HRWLE, it does not rely on special hardware features (e.g., suspend-resume) available only on POWER8 processors by IBM, but assume that the underlying HTM implementation assumes a basic/conventional API for transaction demarcation and, as such, is a generic solution that can be adopted on any HTM system.

As already mentioned, SpRWLock exposes a classic read-write lock interface. As such, SpRWLock can be used as a drop-in, speculative replacement for conventional read-write locks in applications that already use this synchronization primitive; however, it is straightforward to adapt SpRWLock's algorithm to be employed also by TM-based applications, by mapping the begin and commit of read-only and update transactions to lock and unlock requests to a single global lock implemented using SpRWLock.

For the sake of clarity, we present SpRWLock in an incremental fashion. We start by presenting, in Section 3.1 a simple, base algorithm that embodies one of the key ideas at the basis of SpRWLock: enabling safe concurrency between un-

Algorithm 1 — Reader basic algorithm(thread *tid*)

```
1: Gobal variables:
2:     state[N] ← {⊥, ⊥, ..., ⊥}                ▷ One status per thread
3:     gl                                        ▷ global lock for HTM fallback
4: function SpRWL_read_lock
5:     state[tid] ← #READER                      ▷ Flag active reader
6:     mem_fence                                 ▷ Make sure writers see reader
7:     reader_gl_sync()
8: end function
9: function SpRWL_read_unlock
10:    state[tid] ← ⊥                            ▷ Exit critical section
11: end function
12: function reader_gl_sync
13:    if locked(gl) then
14:        state[tid] ← ⊥                        ▷ Defer to gl writer
15:        repeat until !locked(gl)              ▷ wait until lock is free
16:        go to 5
17: end function
```

instrumented readers and HTM-backed writers. We then extend this base algorithm in Section 3.2, by introducing two scheduling techniques that aim both at enhancing performance and ensuring fairness. We conclude by discussing the correctness of the proposed solution (Section 3.3) and presenting a set of relevant optimizations (Section 3.4).

3.1 Base Algorithm

The pseudo-code of SpRWLock base algorithm SpRWLock is reported in two parts: (i) the reader part 1 and (ii) the writer part 2. In the following, for brevity, we will refer to the threads that request to acquire the lock in read/write mode as readers/writers, respectively.

As already mentioned, write critical sections are executed speculatively, using HTM: a write lock acquisition request triggers the activation of a HTM transaction and the corresponding unlock request triggers the commit of its associated hardware transaction. Readers, conversely, are executed un-instrumented, i.e., without recurring to HTM, and are therefore spared from HTM's inherent limitations.

In order to ensure the safety of readers, in presence of concurrent writers exe-

Algorithm 2 — Writer basic algorithm(thread *tid*)

```
1:  Gobal variables:
2:      state[N] ← {⊥, ⊥, ..., ⊥}                    ▷ One status per thread
3:      gl                                            ▷ global lock for HTM fallback
4:  function SpRWL_write_lock
5:      attempts ← 0
6:      state[tid] ← #HTM_WRITER
7:      begin_htm()                                   ▷ Start transaction
8:  end function
9:  function SpRWL_write_unlock
10:     if state[tid] is #HTM_WRITER then
11:         check_for_readers()                       ▷ Abort if there are active readers
12:         tx_commit                                 ▷ Write back updates
13:     else
14:         release_gl()
15: end function
16: function begin_htm
17:     repeat until !locked(gl)                      ▷ wait until lock is free
18:     attempts++
19:     status ← tx_begin()                           ▷ Begin HTM transaction
20:     if status is SUCCESS then
21:         if locked(gl) then                        ▷ Add lock to read-set and...
22:             tx_abort()                            ▷ abort Tx if lock is busy
23:     else
24:         abort_handler()
25: end function
26: function abort_handler
27:     if attempts > MAX_RETRIES then                ▷ is budget over?
28:         state[tid] ← #GL_WRITER
29:         acquire_gl()                              ▷ activate fallback
30:         wait_for_readers()
31:     else
32:         go to 2
33: end function
34: function check_for_readers
35:     for i ← 0 to N−1 do                           ▷ Abort if any thread...
36:         if state[i] is #READER then               ▷ ...is an active reader...
37:             tx_abort()
38: end function
39: function wait_for_readers
40:     for i ← 0 to N−1 do                           ▷ For every thread...
41:         repeat until state[i] ≠ #READER           ▷ wait until it is not an active reader
42: end function
```

cuting in HTM, SpRWLock uses the following mechanism. Before a reader *tid* is granted access to the read critical section, it first advertises its existence to concurrent writers in the *tid*-th entry of the *state* shared array. The update of the state array is followed by a memory fence, which, as we will discuss, is key for correctness,

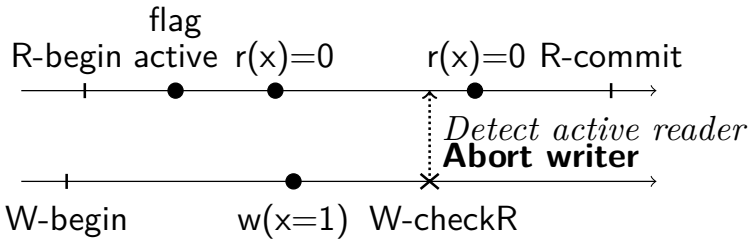

Figure 3.1: A read access during an active update transaction will abort the latter.

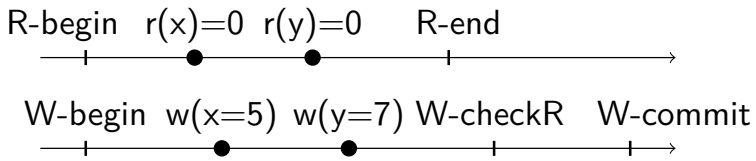

Figure 3.2: A read access which commits before an active update transaction writes on shared values or verifies the state allows it to successfully commit.

as it ensures that the state of readers is globally visible before they enter the read critical section. Upon releasing the read lock, the reader's state is accordingly reset — this time without recurring to memory barriers, though.

Writers, in their turn, check for the existence of concurrent active readers, by inspecting the *state* array, upon requesting to release the write lock, i.e., before attempting to commit the corresponding HTM transaction. Only in case no reader is found active, the HTM transaction can be committed; else, the writer is forcibly aborted and restarted (see Fig.3.1).

This mechanism ensures that no writer can commit and materialize any changes to memory if there is any concurrent, active readers. This, in turn, guarantees that readers execute on isolated snapshots of memory, despite they can run concurrently with HTM-backed writers, as illustrated in Fig. 3.2, as well as with other readers.

In the above description, we have, for simplicity, omitted discussing the management of the fall-back execution path, which, we recall, is required in HTM systems to ensure termination of transactions that cannot be successfully executed in hardware. As in typical HTM systems, SpRWLock uses a *single_global_lock* (SGL)

as fall-back synchronization method: in case a transaction cannot complete successfully in HTM after some predetermined number of attempts, the transaction is executed pessimistically, after having acquired the SGL. SGL is also *subscribed* right after a hardware transaction begins, i.e., the lock's state is read and the transaction is aborted if the lock is not found free. This guarantees that if a thread activates the fall-back path and acquires the SGL, any concurrent hardware transaction is immediately aborted.

In order to ensure the correct interplay between un-instrumented readers and writers active using the SGL, readers check the SGL after flagging their own state to active, and are allowed proceed only if the SGL is found free (see line 13). The writers that execute in the fall-back path, in their turn, have to wait for the completion of any active reader after acquiring the SGL and before executing the write critical section (see line 2). Overall, this mechanism ensures safety by precluding any concurrency between un-instrumented readers and writers executing in the SGL.

As we will show, despite its simplicity, this base algorithm is surprisingly effective in boosting system's throughput in workloads dominated by long readers that do not fit HTM's capacity. Indeed, if one attempted to use plain HTM to elide a read critical section that does not meet the hardware capacity limitations, the reader would eventually exhaust its budget of retries using HTM and acquire the SGL fallback. This would prevent any concurrency with other readers and/or writers. Conversely, with SpRWLock, readers that exceed HTM's capacity can still execute concurrently not only with other readers, but also with other writers executing in HTM, as exemplified by Fig. 3.2.

However, since writers are only allowed to commit using HTM in absence of concurrent readers, in read intensive workloads, this base algorithm exposes writers to the risk of starvation. More precisely, this approach can expose writers to the risk of exhausting their budget of retries in HTM, leading to frequent activations of the pessimistic fall-back path that can hinder not only the latency of writers, but also the global degree of concurrency in the system.

3.2 Scheduling Techniques

In order to address the above discussed shortcomings, SpRWLock integrates two additional scheduling techniques, which we refer to as reader and writer synchronization schemes. The former imposes delays on the readers' side, in case they detect active writers, whereas the latter imposes delays to writers, if they detect active readers. The two synchronization schemes operate in a synergistic fashion, ultimately aimed to enhance SpRWLock's efficiency, but they do pursue different goals.

Specifically, the reader synchronization scheme pursues a twofold goal: i) providing fairness guarantees for the writers, by ensuring that newly readers cannot cause the abort of already active writers, and ii) reducing the. The writer synchronization scheme, conversely, stalls writers to prevent them from uselessly consuming their budget of attempts using HTM, while striving to achieve maximum concurrency with any active reader.

Also, in this case, we present the two techniques in an incremental fashion, introducing first the reader synchronization scheme and then the writer synchronization mechanism.

3.2.1 Reader Synchronization

The pseudo-code of the reader synchronization scheme is reported in Alg. 3. Note that the pseudo-code illustrates only the differences with respect to the base algorithm, omitting the parts in common. This variant uses two additional shared arrays, also having one entry per each thread in the system: $clock_w$, which stores the expected end time of any currently active writer, and $waiting\_for$, which is used by readers to advertise the identity of any writer they are currently waiting for.

In order to estimate the expected end time of (write) critical sections in a lightweight, yet accurate, fashion, SpRWLock relies on the hardware time stamp counter, which in modern CPUs provides a low-overhead, cycle-accurate time source.

Algorithm 3 — Reader synchronization algorithm (thread *tid*)

```
1: Shared variables:
2:     ...                                                  ▷ Similar to Alg. 1
3:     clock_w[N] ← {⊥, ⊥, ..., ⊥}                          ▷ per thread
4:     waiting_for[N] ← {⊥, ⊥, ..., ⊥}                      ▷ per thread
5: function SpRWL_write_lock
6:     attempts ← 0
7:     state[tid] ← #HTM_WRITER
8:     clock_w[N] ← expected_end                            ▷ Publish expected end time
9:     begin_htm()                                          ▷ Start transaction
10: end function
11: function SpRWL_read_lock
12:     readers_wait_function()                              ▷ Sync with active writers
13:     state[tid] ← #READER                                 ▷ Enter critical section
14:     mem_fence                                            ▷ Make sure writers see reader
15:     reader_gl_sync()                                     ▷ Enter critical section
16: end function
17: function readers_wait_function
18:     wait ← false
19:     max_wait ← 0
20:     for i ← 0 to N−1 do                                  ▷ Wait if there are...
21:         if state[i] is #WRITER and clock_w[i] > max_wait then   ▷ ...active writers...
22:             max_wait ← clock_w[i] rdtsc()                ▷ ...for the most lasting ...
23:             wait ← i
24:         else if waiting_for[i] ≠ ⊥ then                  ▷ ...or wait with a...
25:             wait ← waiting_for[i]                        ▷ ...waiting reader.
26:             break
27:     waiting_for[tid]← wait
28:     repeat until state[wait]≠ #WRITER nop                ▷ Wait until most lasting writer finishes.
29:     waiting_for[tid] ← ⊥
30: end function
```

Further, in order to cope with programs having critical section of heterogeneous duration, SpRWLock gathers independent statistics for different critical sections[1]. More in detail, SpRWLock samples the execution time of critical sections on a single thread[2] — so as to reduce measurement overhead — and computes an exponential moving average — which can be efficiently computed in an on-line fashion and

[1]In order to identify different critical sections, the current prototype of SpRWLock requires programmers to specify a unique identifier to the APIs used to enter/exit a critical section. This task could, however, be delegated to the compiler and made fully automatic and transparent for programmers, using, e.g., the stack trace as unique identifier of different critical sections

[2]This approach assumes that the sampling thread eventually executes all the critical sections that can ever be acquired by *any* thread. It also assumes that the measurements gathered by the sampling thread are representative for every other thread. These assumptions hold for all the applications that we have tested in our experimental study, but may not be true for applications where threads play specialized/heterogeneous roles. These scenarios could be accommodated by sampling the duration of critical sections at multiple threads, and periodically merging the statistics gathered at each thread, as, e.g., in [39].

allows for quickly reflecting changes in the workload characteristics. To simplify presentation, we omit describing explicitly these mechanisms in the pseudo-code and encapsulate them in the *estimateEndTime()* primitive.

The reader synchronization mechanism introduces two main changes to the base algorithm presented in Section 3.1.

First, upon requesting a write critical section, writers advertise their existence and expected end time in the *state* and $clock_w$ arrays, respectively.

Second, before entering a read critical section (via the SpRWL^{basic}_read_lock() function), readers check whether they have to first execute a wait phase (Readers_Wait() function). More in detail, a reader inspects the *state* and $waiting\_for$ arrays' entries of the other threads in the system and starts a waiting phase in case i) it finds any active writer, or ii) any reader already waiting for an active writer.

If there are no readers already waiting, the newly arrived reader waits for the writer that is expected to complete last. It is easy to see that this ensures that newly arrived readers do not prevent already active writers from committing using HTM.

If, instead, a newly arrived reader r detects the existence of another reader r' waiting for some writer w, r joins r' in the wait for w. As we will show experimentally, this policy has a relevant beneficial impact on performance at high thread counts, since it tends to synchronize the starting time of readers. If readers are likely to begin simultaneously, the time window there is any active reader in the system is globally reduced, increasing the chances for writers to be able to execute using HTM. Further, it tends to reduce the average reader latency, by allowing them to start sooner.

3.2.2 Writer Synchronization

The writer synchronization scheme, whose pseudo-code is reported in Alg. 4, aims at sheltering writers from the risk of incurring repeated aborts due to already active readers, while striving to jointly maximize the concurrency achievable by writers

Algorithm 4 — Writer synchronization algorithm (thread *tid*)

```
1: Shared variables:
2:     ...                                              ▷ Similar to the Alg. 3
3:     clock_r[N] ← {⊥, ⊥, ..., ⊥}                      ▷ per thread
4: function SpRWL_read_lock
5:     reader_wait_function()                           ▷ Sync with active writers
6:     clock[tid] ← expected_end
7:     state[tid] ← #READER                             ▷ Enter critical section
8:     mem_fence                                        ▷ Make sure writers see reader
9:     reader_gl_sync()                                 ▷ Enter critical section
10: end function
11: function abort_handler
12:     attempts++
13:     if attempts > budget then                        ▷ Ran out of budget?...
14:         ...                                          ▷ Similar to the Alg. 2
15:     else
16:         if abort_cause is reader_abort then
17:             writer_wait_function()                   ▷ Sync with active readers
18:         ...                                          ▷ Similar to the Alg. 2
19: end function
20: function writer_wait_function
21:     wait ← 0
22:     for i ← 0 to N−1 do                              ▷ For each thread...
23:         if state[i] is #READER then                  ▷ ...check for the...
24:             if clock_r[i] > wait then                ▷ ...most lasting reader.
25:                 wait ← clock[i]
26:     wait −= length + overlapping_offset
                ▷ Start the writer as soon as possible, while overlapping with readers
27:     repeat until rdtsc() ≥ wait
28: end function
```

and readers.

In a nutshell, these goals are pursued by delaying the starting time of a writer that executes in HTM, in case it encounters any active reader, by the shortest time possible that still allows the writer to commit successfully. This is achieved by timing the start of the writer so that the execution of the corresponding write critical section completes "shortly after" the last reader. This way, not only writers are guaranteed (or at least is more likely) not to have to abort due to a concurrent reader, they can also maximize the period of time during which their execution overlaps with concurrent readers.

More in detail, a writer first attempts, optimistically, to execute in HTM immediately, i.e., avoiding the writer synchronization phase. Note that the pseudo-code

in Alg. 4 only reports the parts that differ with respect to Alg. 3 and, as such, omit specifying the pseudo-code for entering a write critical section, which is not modified by the writer synchronization scheme.

If a writer, executing in a HTM transaction, incurs an abort, it determines the maximum end time of any active reader. To this end, with the writer synchronization scheme, also readers advertise their expected end time, right after having completed the reader synchronization and before starting execution (see function SpRWL^{wSync}_READ_LOCK) using a dedicated global array ($clock\_r$). In order to overlap its execution with that of already active readers, a writer adjusts its waiting phase so that it is expected to complete δ cycles after the last active reader. δ is a parameter, whose tuning allows to trade-off the degree of concurrency between writers and readers (which can be increased by setting δ close to 0) for the likelihood that writers can commit successfully (which can be increased by setting δ close to the expected duration of the writer). In SpRWLock, we use half the expected duration of the writers as default value for δ, which we have observed to provide the best over-all performance based on preliminary experimentations.

It should be noted that, to preserve fairness, writers maintain their writer flag active while waiting for a reader: this guarantees that writers have a chance to commit successfully, as new coming readers will wait for them thanks to the reader synchronization scheme.

3.3 Correctness and Fairness

This section elaborates on the correctness and fairness guarantees ensured by SpRWLock.

As in typical HTM systems, the correctness (more precisely safety) criterion ensured by SpRWLock is opacity [40]. We analyze separately the case of readers and writers.

The correctness of readers is ensured by two mechanisms: *(i)* the check performed by writers using HTM before committing, *(ii)* the wait performed by writers executing in the SGL path before they begin. Recall that HTM guarantees that

memory writes issued in a HTM transaction are hidden from concurrent threads during transaction's execution and will appear atomically only upon a successful commit. Therefore, for a HTM writer to break the consistency of a reader, it must commit throughout the course of the reader's execution (Fig. 1.1). However, this is impossible, since readers advertise their presence using a memory barrier before starting, which guarantees that any writer using HTM will detect the reader's presence before committing and abort.

For analogous reasons, writers that activate the fallback path are guaranteed to detect and wait for any active reader before starting executing (line 2, Alg. 2). Further, readers start after a writer has acquired the SGL are guaranteed to detect the writer's presence and are forced to wait for its completion (line 13, Alg. 1). Overall, these two synchronization mechanisms prevent any concurrency between an un-instrumented reader and writers executing in the fallback path.

Let us now discuss the liveness and fairness properties of SpRWLock. We note that the synchronization scheme employed by SpRWLock to avoid concurrency between un-instrumented readers and writers executing in the SGL cannot lead to deadlocks. In fact, in order for a writer to wait for a reader (line 2, Alg. 1), the reader must be advertising its state as active. However, before starting waiting for any writer (line 14, Alg. 1), readers first reset their state flag (line 15, Alg. 1). This precludes the possibility of mutual waits/deadlocks between writers and readers.

Further, since writers wait for readers only after having acquired the SGL, and they wait for a given reader at most once, SpRWLock guarantees that a writer that activates the fallback path cannot wait indefinitely (line 2, Alg. 2), even in presence of a constant stream of readers. We note that SpRWLock algorithm may, theoretically, cause readers to wait indefinitely in presence of a constant stream of writers executing in the SGL. This issue can be avoided by implementing the SGL via a versioned lock that is incremented each time writers acquire it. The first time a reader finds the SGL busy, it registers and advertise in a per-thread variable the current lock's version number and only waits for writers if the value is

not larger than the one the reader first observed. Writers, in their turn, acquire the SGL and increase the lock's version, but start executing only if there are no readers waiting with a smaller version number. We omitted this optimization since in read-dominated workloads, i.e., the ones targeted by read-write locks and SpRWLock, the probability for a reader to starve due to a continuous stream of writers is expected to be quite low.

Finally, as already mentioned, the base version of SpRWLock algorithm can cause writers that execute in HTM to be overrun by newly arrived readers. This leads to a violation of fairness for HTM-backed writers, which can lead to frequent and unnecessary activations of the fallback path. This issue is tackled by the reader synchronization scheme, which guarantees that if a reader r starts after a writer w, r will only start after w completes executing.

3.4 Optimizations

This section describes a set of relevant performance-oriented optimizations that we integrated in SpRWLock, but omitted while presenting its pseudo-code, to simplify presentation.

A second optimization consists in employing the Scalable NonZero Indicators (SNZI) [41] to allow writers to detect the existence of concurrent readers, at commit time. The previously presented pseudo-code, in fact, forces the writers to incur a cost linear in the number of threads in the system, given that it requires writers to scan the entire *state* array. Since the reading of the *state* array takes place from within the scope of a HTM transaction, this results in an increase of the memory footprint of the encompassing hardware transaction and in a reduction of the cache capacity effectively available for writers. Also, the longer it takes for a writer to scan the *state* array, the more it is exposed to the risk of aborting due to a conflict with a concurrent reader that changes its own entry. The two issues above can be avoided by having readers advertise their existence via SNZI, since SNZI can execute queries in constant time, by reading a single counter. This comes, though, at the cost of

an increase in the overhead incurred by readers to advertise the start/end of their execution, as updates to SNZI have, roughly, logarithmic complexity in the number of threads.

Chapter 4

Evaluation

In this chapter we evaluate SpRWLock against a number of RWL implementations that use either speculative or pessimistic techniques. The experimental study is conducted on two HTM-enabled processor (by Intel and IBM) characterized by different capacity limitations, and encompasses a large set of synthetic and complex benchmarks/real-life applications.

More in detail, we consider the following baseline solutions: SpRWLock is evaluated against: *(i)* pthread's RWL (rwl), *(ii)* Big Reader Lock [42] (brlock), *(iii)* plain transactional lock elision (tle) and *(iv)* hardware read-write lock elision (herwl) [43], which, unlike SpRWLock, relies on specific features of IBM POWER8 processor (see Section 2.4.2) and, as such, cannot be tested on Intel platforms.

For all HTM based solutions, including SpRWLock, we used a retry policy that attempts a transaction 10 times in hardware before activating the fallback path except in case of capacity aborts, where a transactions is directly executed using the fallback path. The only exception is HRWLE, where we used a budget of attempts equal to 10 for update transactions executing using ROTs — the same policy used by the authors of HRWLE in their evaluation. Previous works have shown that dynamically tuning the budget of HTM retries can lead to performance gains in some workloads [44], but in our study we use a common, static retry policy to simplify the analysis of the results.

The evaluation is performed using two different architectures that provide support for HTM, namely Intel Broadwell and IBM POWER8. For Intel, we used a dual socket Intel Xeon E5-2648L v4 processors with 28 cores and up to 56 hardware threads running Ubuntu 16.04 with Linux 4.4. For IBM, we used a POWER8 8284-22A processor that has 10 physical cores, with 8 hardware threads each running Fedora 24 with Linux 4.7.4. It should be noted that, due to their architectural differences, these Intel's and IBM's processors are faced with very different capacity limitations. Specifically, POWER8 processors have a 8KB capacity for both memory reads and writes, whereas the capacity of Broadwell is 22KB for writes and 4MB for reads [45].

The source code, which will be made public [46], was compiled with -O2 flag using GCC 5.4.0 and 6.2.1 for the Intel and IBM platforms, respectively. Thread pinning was use to pin a thread per core at the beginning of each run for all the solutions, and threads were distributed evenly across the available CPUs. All the results reported in the following are obtained as the average of at least 5 runs.

The remainder of this section is structured as follows.

First, we conduct as sensitivity analysis aimed to stress different aspects of SpRWLocks design and optimizations, such as: the relevance of the reader and writer synchronization schemes and using SNZI vs per thread flags to track the status of active readers. To this end, we rely on a micro-benchmark, based on a concurrent hashmap, which allows us to control in a precise way the workload's characteristics.

Finally, we test SpRWLock using two complex benchmarks and a real-life application: STMbench7 [47], a port of TPC-C [13] for in-memory databases [48, 49] and KyotoCabinet [15].

4.1 Sensitivity Analysis

In this section we describe the results of a sensitivity analysis based on a micro benchmark consisting of a hashmap that offers three operations: lookup, insert

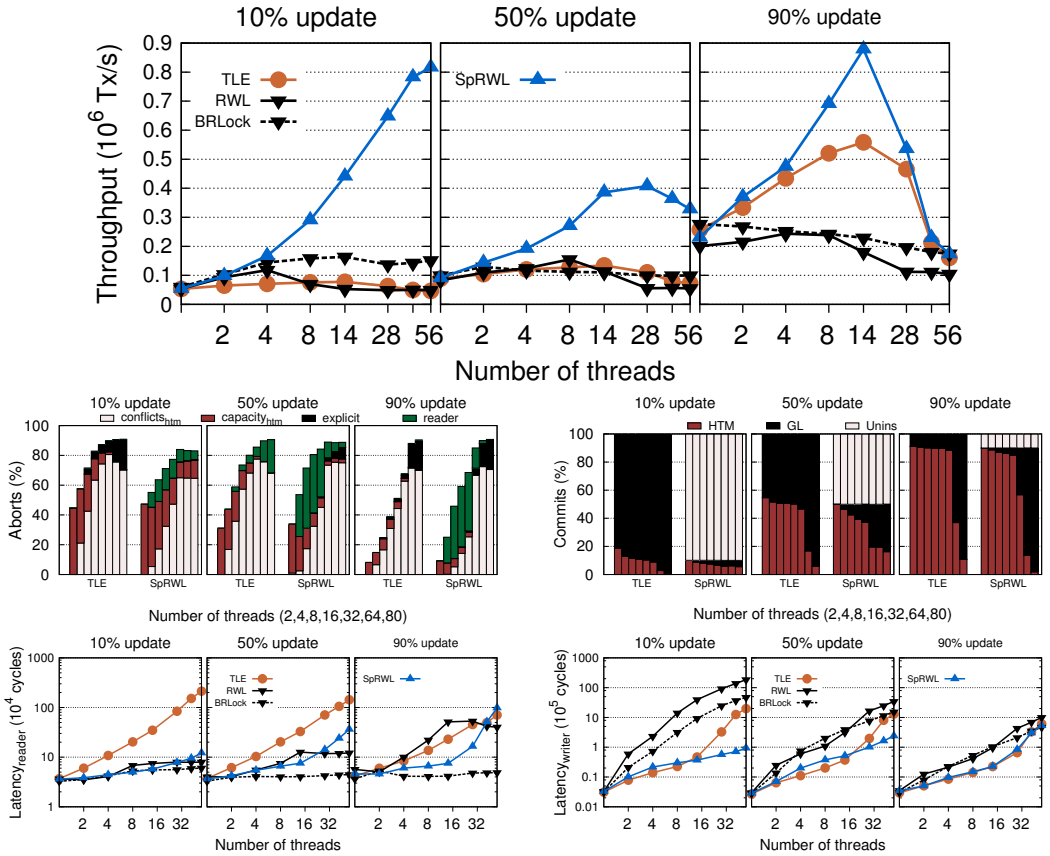

Figure 4.1: Hashmap: reader's size = 10 × writer's size configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios on Intel.

and delete. The ratio of lookups to inserts and deletes controls the percentage of update transactions performed. We protect each insert and delete operation in a write lock and pre-populate the hashmap so that update operations fit the capacity limitations of the underlying HTM implementation. To this end, we set the number of hashtable's buckets to 5000 and populate the data structure with 8 million and 3 million items for the Broadwell and Power8 processor, respectively.

We control the size of readers by performing either 1 or 10 lookups within a single read critical section. In the latter case, lookups exceed in the HTM capacity, which allows for quantifying the gains that SpRWLock can achieve with respect to both HTM-based solutions as well as pessimistic ones. In the former case, in-

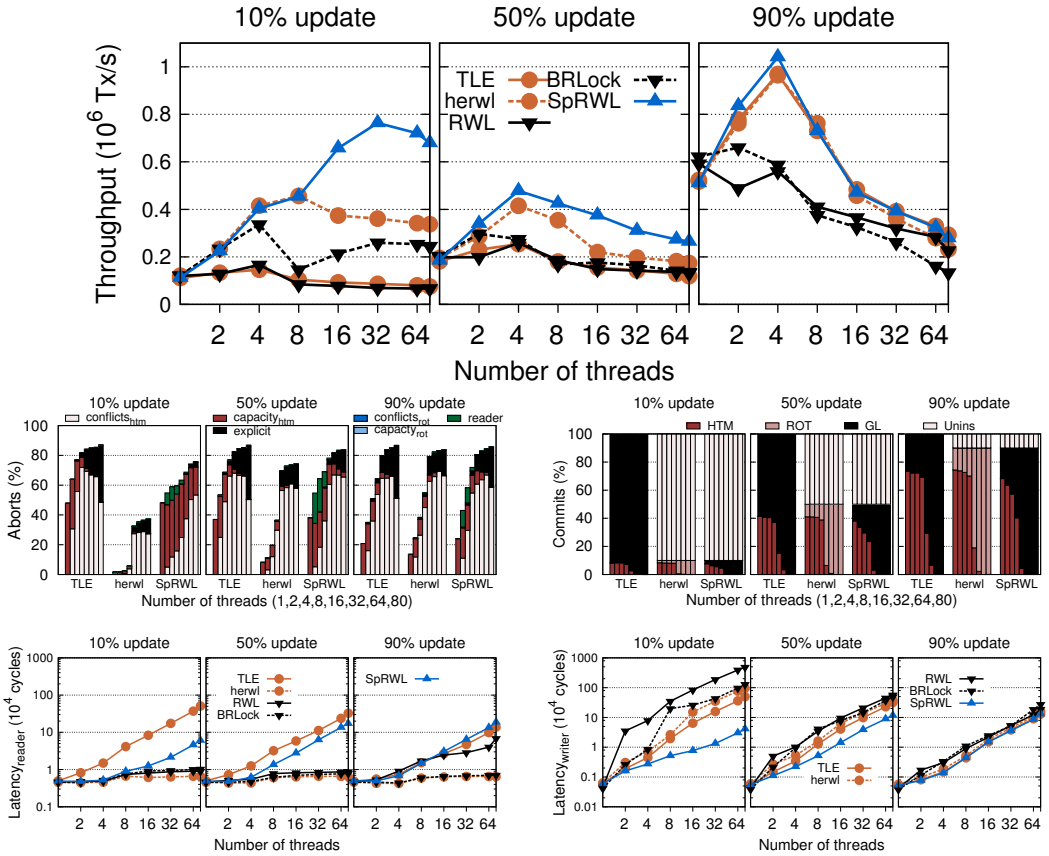

Figure 4.2: Hashmap: reader's size = 10 × writer's size configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios on POWER8.

stead, lookups can be successfully executed in HTM, which nullifies the benefits of SpRWLock with respect to plain HTM-based lock-elision — thus, allowing us to evaluate SpRWLock's overhead in an unfavorable workload.

In this section, when running SpRWLock, we enable all the scheduling techniques and optimizations presented in Section 3, with the exception of the SNZI-based readers' tracking mechanism, which will be investigated in detail in Section 4.1.2.

In Figures 4.1 and 4.2 and , we show the results for executing readers that perform ten lookup operations within a single read critical section. On the left we report data for Broadwell and on the right for Power8. From top to bottom, for each architecture, we report the throughput, abort rate, breakdown of commit

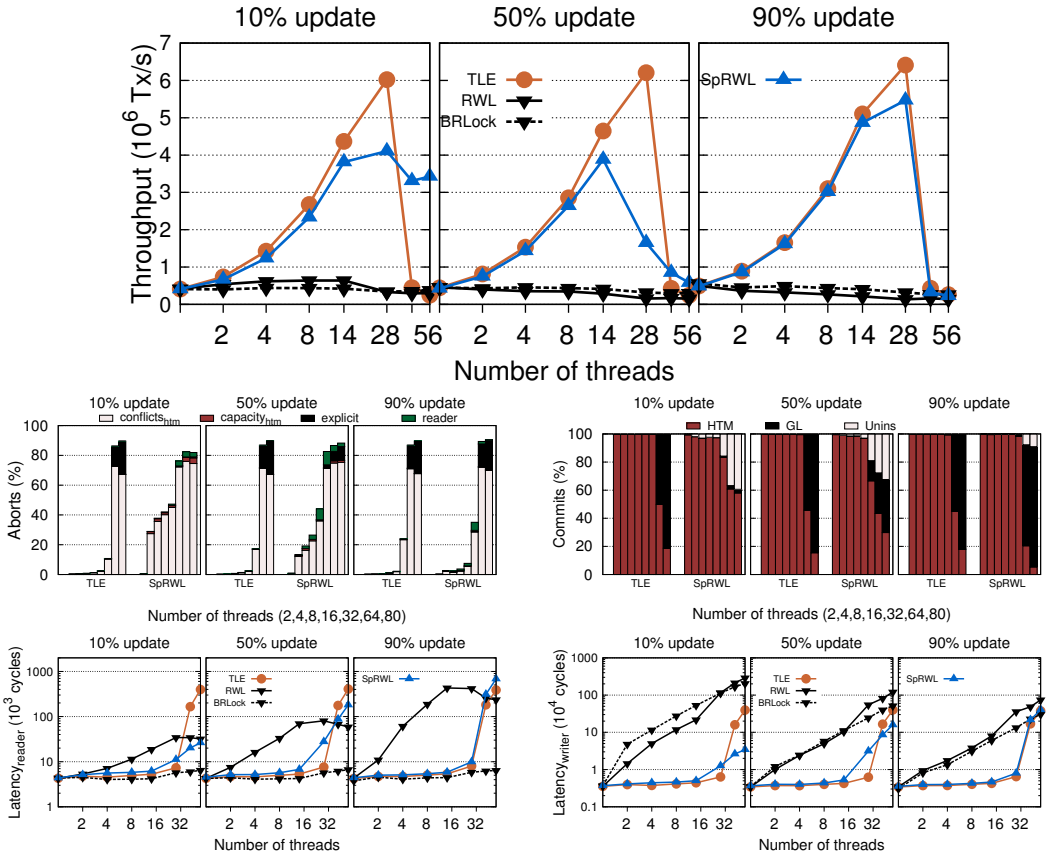

Figure 4.3: Hashmap: reader's size = 1 × writer's size configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios on Intel.

modes, as well as the readers' and writers' latency. From left to right, we increase the percentage of update operations/critical sections from 10% to 50% and 90

As mentioned, in this workload, readers do not normally fit in HTM transactions, as confirmed by the aborts breakdown which shows almost 50% capacity abort rate with a single thread on both Broadwell and POWER8. The high percentage of capacity aborts due to large readers leads to the frequent activation of the pessimistic fallback path, as reflected in the commit breakdown plot. Consequently, Transactional Lock Elision (TLE) achieves modest scalability in both the 10% and 50% update workloads.

Thanks to its ability to execute readers without instrumentation (see com-

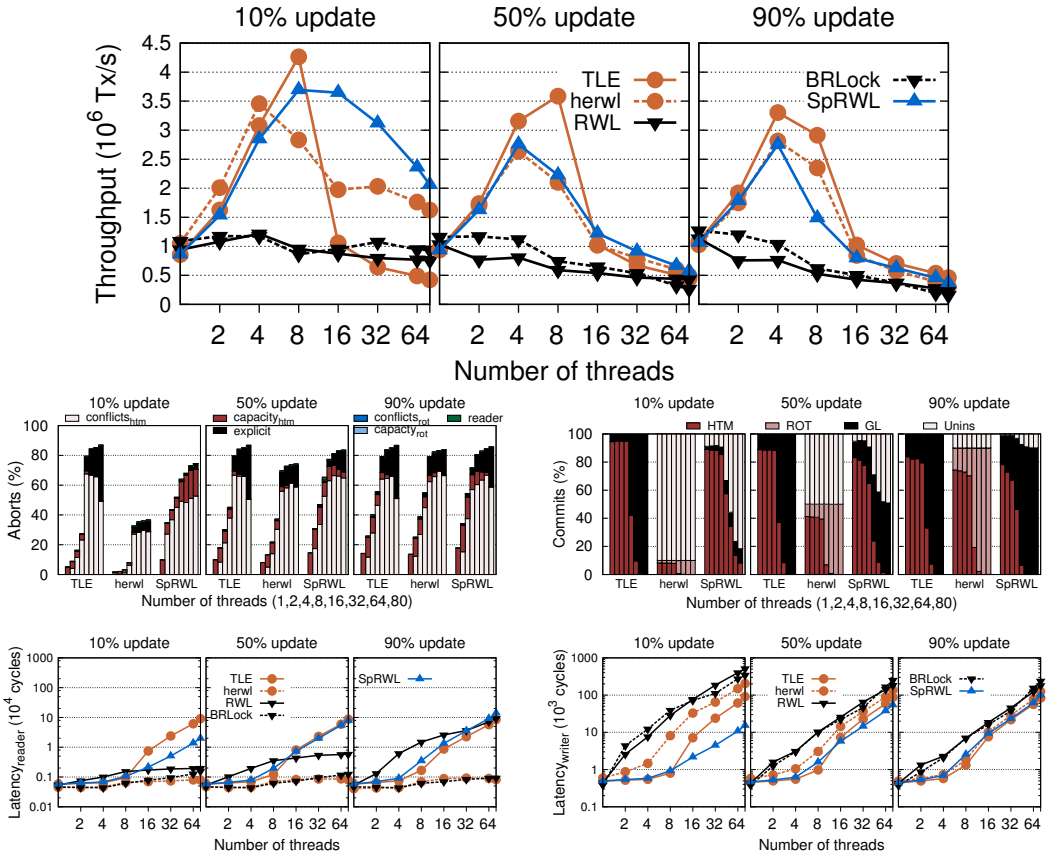

Figure 4.4: Hashmap: reader's size = 1 × writer's size configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios onPOWER8.

mit breakdown plot), instead, SpRWLock exhibits a much better scalability level, achieving, in the 10% updates workloads, speedups versus TLE up to 16×/8× on Broadwell/POWER8, respectively. When moving to workloads with higher percentage of update operations — which fit HTM's capacity — we notice that SpRWLock still outperforms TLE by up to 5×/2× on Broadwell/POWER8 in the 50% workload. Even in a 90% workloads, SpRWLock is still capable of yielding 2× higher throughput than TLE on Broadwell and marginally higher throughput on POWER8. The ability of SpRWLock to scale more on Broadwell as compared with POWER8 is related to the fact that our Broadwell machine supports up to 28 threads without hyperthreading; conversely, when using more than 10 threads,

on POWER8, multiple (up to 8) hardware threads start sharing the same physical cores, reducing their effective capacity, and, accordingly, their ability to execute write critical sections concurrently, using HTM.

When compared with pessimistic solutions, namely BRLock and RWL, similar trends are observe: SpRWLock achieves up to 4×/2.5× throughput gains compared to the best performing of the two (BRLock) on Broadwell/POWER8 in the 10% update workloads. Furthermore, in workloads with higher percentage of update operations, specially the 90% workload, we can see that SpRWLock achieves higher gains, due to its ability to execute writers concurrently using HTM.

HRWLE, which also executes readers without instrumentation on top POWER8, is able to achieve similar performance to SpRWLock up to 8/4 threads in the 10/50% updates workloads. However, it does not scale beyond that point. This behavior can be explained by the commits breakdown, which shows that HRWLE execute all update transactions as ROTs from that point on. Although ROTs allow concurrent readers, unlike the pessimistic lock used by SpRWLock, before ROTs can be committed writers need to execute a quiescence phase to wait for the completion of any active reader. In a workload with long readers, this leads writers to incur significant overheads, as highlighted by the writers' latency plot. By executing writers in the SGL, SpRWLock avoids this issue, reducing the average writers' latency by more than 2 orders of magnitude lower latency at 32 threads — the point where SpRWLock achieves the highest throughput gains with respect to HRWLE — at the cost of a (relatively) much smaller increase of the readers' latency (approximately 3×).

Figures 4.3 and 4.4 show the throughput and breakdown of aborts and commits when readers perform a single lookup operation in the read lock, thus fitting in HTM transactions. As expected, since also update operations can be successfully executed using hardware transactions, TLE is the overall best performing variant, achieving the highest throughput across all workloads and architectures. Indeed, by looking at the commits breakdown, we can not notice the ability of TLE to commit

almost all transactions using HTM, except for very high thread counts (due to the coexistence of multiple hardware threads on the same physical core).

Nevertheless, even in this unfavourable workload, SpRWLock achieves performance comparable with TLE. TLE does attain highest peak throughputs, up to 30% higher than SpRWLock (on Intel 50% update), as it avoids the overheads of the additional, software-based, synchronization mechanisms that SpRWLock employs. However, the average throughput across all thread counts of the two solutions is, on average, 36% higher for SpRWLock. At high thread counts and read-dominated workloads, in fact, readers using TLE start incurring capacity exceptions, which increases the frequency of acquisition of the pessimistic fall-back path and cripples performance. A problem that SpRWLock avoids by allowing readers to execute concurrently and uninstrumented. Further, at low thread counts, SpRWLock can commit a percentage of HTM transactions similar to the one achieved by TLE, avoiding most of the overheads it incurs to support the safe execution of uninstrumented readers. This is possible thank to SpRWLock's policy, presented in Section 3.4, which first attempts, in an optimistic fashion, to execute readers using HTM.

HRWLE, which is the other hardware-based system that executes readers without instrumentation, follows trends similar to SpRWLock. However, it pays a higher penalty in terms of performance when compared with either SpRWLock or TLE. Even though HRWLE always commits update critical sections either as HTM or ROTs, the fact that ROTs are serialized and both HTM and ROT have to wait for active readers (that can fit in HTM) limit its scalability.

When comparing with the pessimistic variants, RWL and BRLock, we can clearly see their inability to scale. This can be mainly related to the high cost they impose relative to the small size of the critical sections they protect.

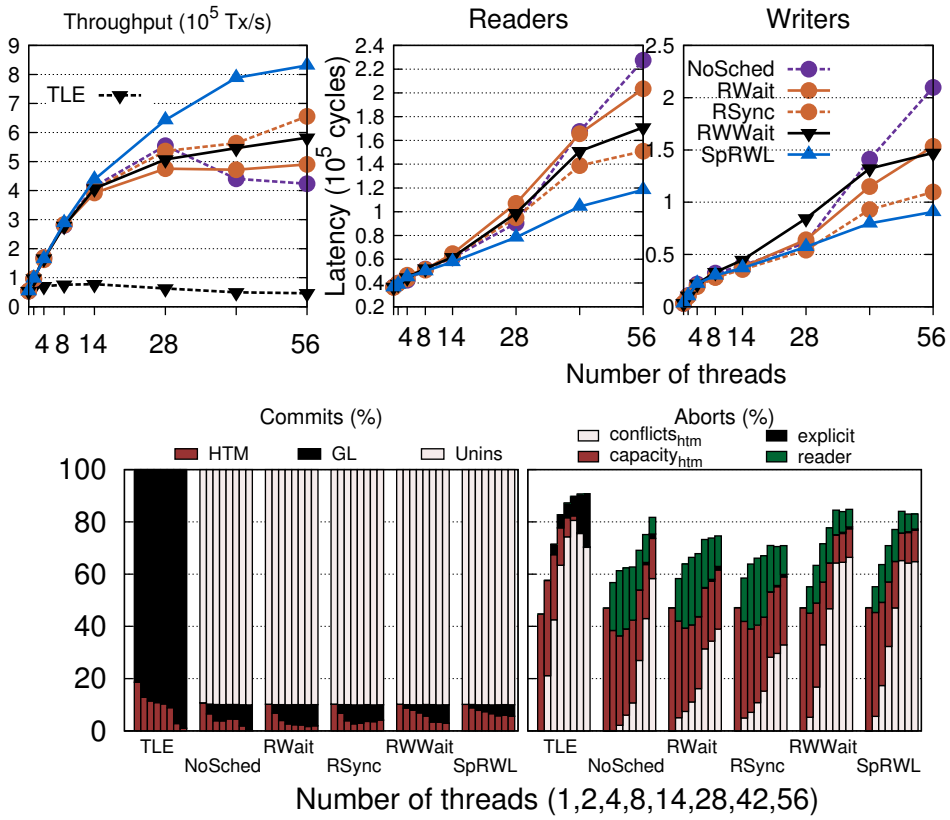

Figure 4.5: SpRWLock variants: readers execute 10 lookups, writers execute 1 insert/delete. 50% update operations on Intel.

4.1.1 Impact of scheduling

In this section we aim at assess the impact on performance of the different scheduling techniques employed by SpRWLock. To this end we developed several variants of SpRWLock, which we obtained by selectively disabling different scheduling mechanisms:

- NoSched : the base version of SpRWLock, which does not employ any scheduling technique, described in Section 3.1.
- RSync : the version of SpRWLock presented in Section 3.2.1, in which readers wait for the active writer that is predicted to complete last, if no readers are already waiting for a writer, and that otherwise makes newly arrived readers

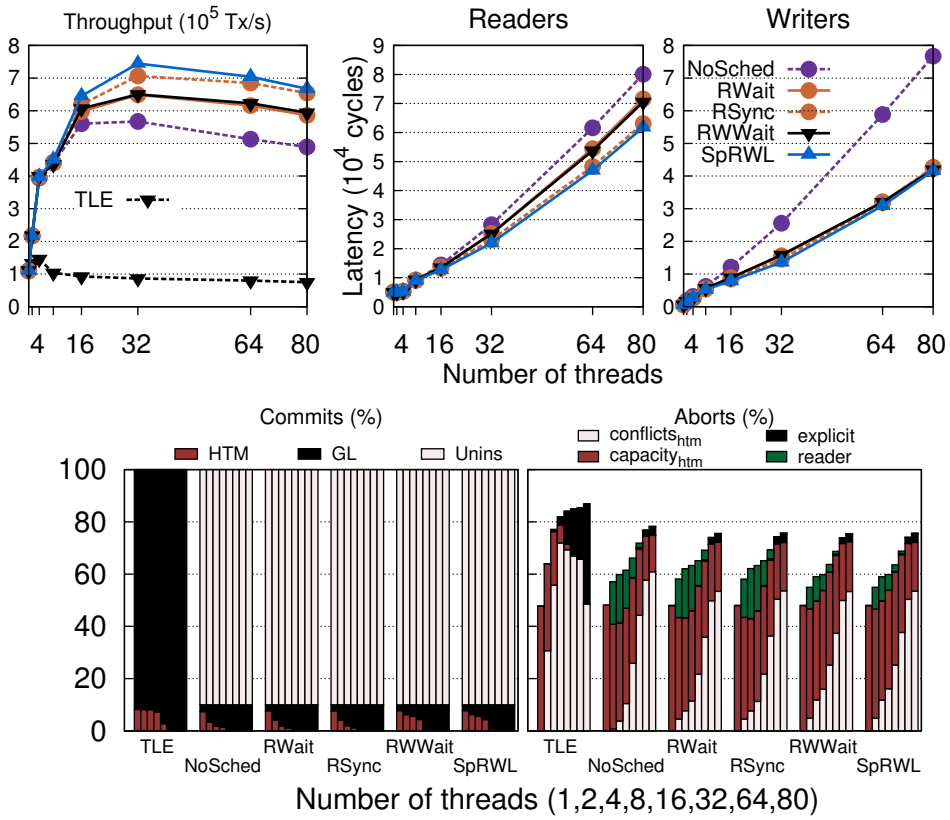

Figure 4.6: SpRWLock variants: readers execute 10 lookups, writers execute 1 insert/delete. 50% update operations on POWER8.

join already waiting ones.

- RWait : A variant of Rsync, in which readers simply wait for the active writer that is predicted to complete last, but do not join other awaiting readers;

For this evaluation we used the hashmap microbenchmark, with the same geometry and initial population as in the previous study.The workload requests with 50% probability write critical section, which execute an individual update operation. 10 lookup operations are instead executed in the read critical sections.

The results obtained (Figs.4.5 and 4.6) show that even the base version of SpRWLock shows significant throughput gains with respect to HTM. However, at high thread counts, the amount of concurrent read transactions increases, reducing

the time window during which updates can commit. As this happens, update transactions fall-back more frequently, which not only hinders their latency but also the overall system's throughput.

The RWait policy brings some noticeable gains, both in terms of throughput and writer latency, at high thread counts. This is due to the fact that update transactions, executed in HTM, are no longer overrun by newly arriving readers, which now wait for already active updates to commit before starting.

RSync shows improvements compared to RWait since, as mentioned in 3.2, it allows read transactions to begin sooner, reducing reader latency, and to synchronize their start time, narrowing the time window during which update transactions are forced to abort (due to the existence of some active reader). This reflects into a reduction of the abort rates and a decrease of the writer latency compared to RWait, and, ultimately, into up to 30% throughput gains.

Finally, the comparison of RSync and SpRWLock allows us to appreciate the additional gains of the writer synchronization scheme, which, especially on Intel, reduces significantly the aborts incurred by writers due to concurrent readers, enhancing the likelihood that they commit successfully in HTM and yielding a further improvement of the peak throughput by an additional 30%. Overall, this study provides quantitative evidence on the synergistic contribution and actual relevance of the scheduling techniques that SpRWLock employs.

4.1.2 Reader tracking scheme

We now evaluate the effects of employing the SNZI-based reader tracking mechanism, described in Section 3.4. The use of SNZI allows writers to determine in a much more efficient way (a single memory lookup) whether there is any concurrent active reader, sparing them from inspecting the *state* array, whose size grows linearly with the number of threads in the system. This comes, however, at the cost of an increased overhead for readers to advertise their presence, which grows logarithmically with the number of threads in the system.

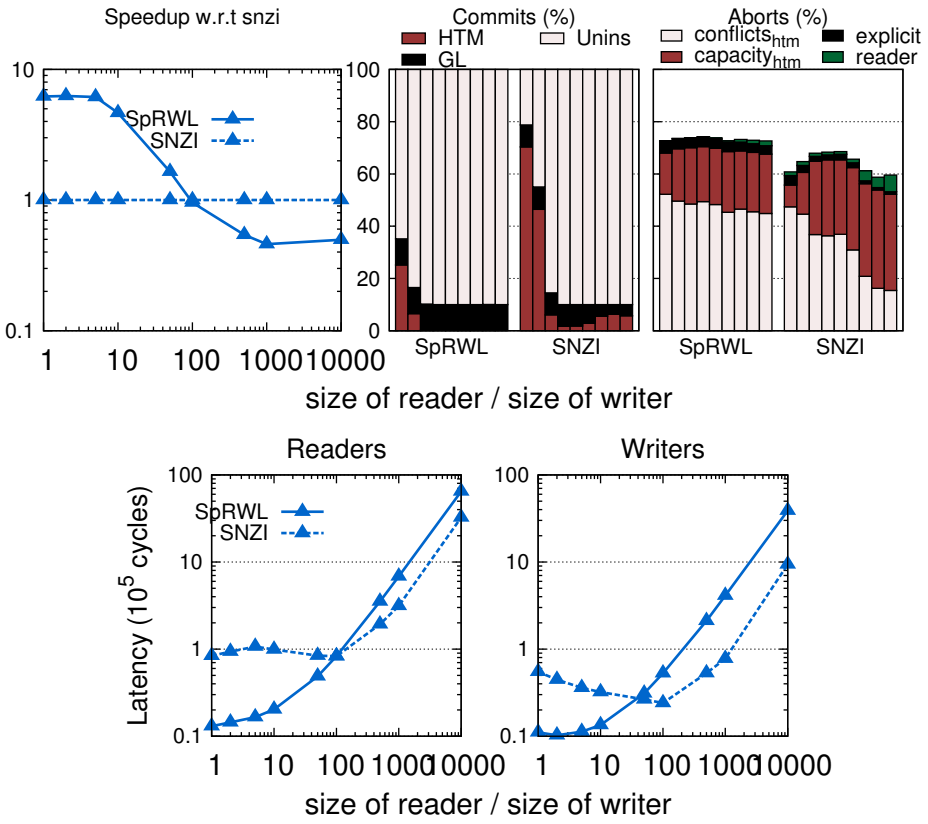

Figure 4.7: Reader tracking scheme: Hashmap, 10% update operations, while varying the size of readers at 80 threads on POWER8.

In order to evaluate this trade-off, we vary the size of the read critical sections by increasing the number of lookup operations they encompass, generating a workload with 50% updates. Figure 4.7 reports the results obtained using 80 threads on POWER8. The experimental data shows that, indeed, the use of SNZI can yield significant throughput gains, up to $\tilde{6}\times$, with long readers, but it also imposes large overheads, up to $6\times$ with short readers. This is a direct consequence of the larger overhead that SNZI imposes to readers, which gets effectively amortized by long readers, but plays a dominant role with short readers.

It is interesting to observe also that, for large readers, not only is the writer latency reduced (as expected, since writers incur less overhead to verify the existence of readers), but also the reader latency is smaller, despite reader incur higher costs

with SNZI. This is explicable considering that the reader synchronization forces readers to wait for active writers. Thus, the reduction of the writer latency enabled by SNZI benefits, indirectly, also the readers' latency, by reducing the average time readers spend waiting for writers.

Overall, these experimental data suggests that the effectiveness of the SNZI-based reader tracking scheme is strongly workload dependent, and that there seems to exist a strong correlation with the size of the readers. We argue that this finding could be leveraged to build self-tuning mechanisms aimed at automatically identifying whether to activate or not this mechanism.

4.2 STMBench7

STMBench7 [14] simulates a CAD-like complex applications, composed of several different and large datastructures, 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. We configured STMBench7 to generate a mix of 10 operations: 5 read-only operations that normally exceed HTM capacity, and 5 update operations that normally fit HTM capacity on Intel, at least at low thread counts, but never do in POWER8. We consider three workloads comprising 1%, 10% and 50% update operations. We report the results obtained with this benchmark in Figures 4.8 and 4.9, which includes the performance of glsSYS both without and with the SNZI-based reader tracking scheme, denoted as glsSYS and SNZI, respectively.

Overall, the workload where the two glsSYS variants shine most is the one with 1% of updates, where they achieve throughput gains of 3×/4× on Broadwell/POWER8, respectively, with respect to the best performing baseline, i.e., BRLock. Latency wise, the largest benefits of SpRWLock with respect to BRLock are for writers, whose latency is more than 2 orders of magnitude lower for writers on both architectures. 2×/1.5×, lower on Broadwell and POWER8. Interestingly, the reader latency of glsSYS on Broadwell is on par with the best baseline (BR-

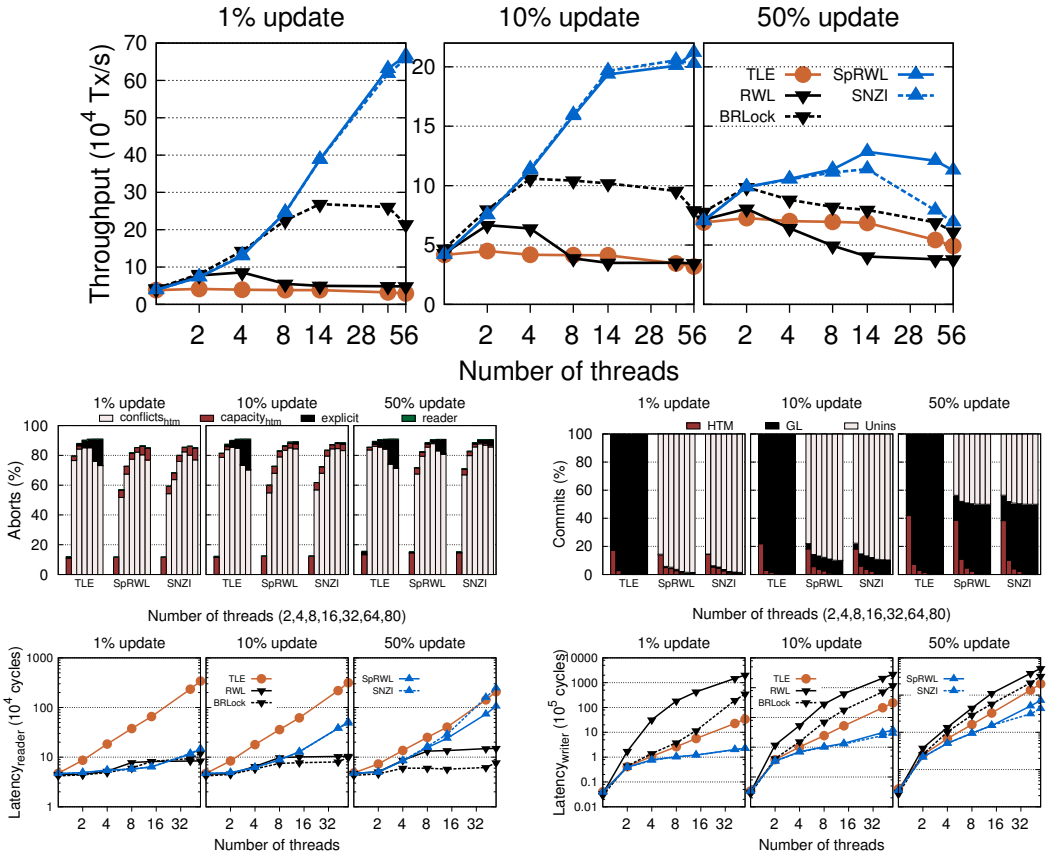

Figure 4.8: STMBench7: throughput, abort rate, and breakdown of commit modes at 1%, 10% and 50% update ratios on Intel.

Lock), but is 10× higher on POWER8. This can explained by considering that in POWER8, which has a smaller capacity than Broadwell, writers never successfully execute in HTM (see the commit breakdown plot). This leads writers to execute sequentially in the fallback path, which translates into an increase of their execution time and, consequently, into an increase of the average wait time imposed to readers due to the reader synchronization scheme. The relatively larger capacity of the Broadwell processors explains also why SpRWLock scales up to 56 threads, but only up to 32 threads on POWER8. The second best alternative,BRLock, instead, only scales up to around 14 threads on both architectures.

As the percentage of update operations grows to 10% and 50%, especially at

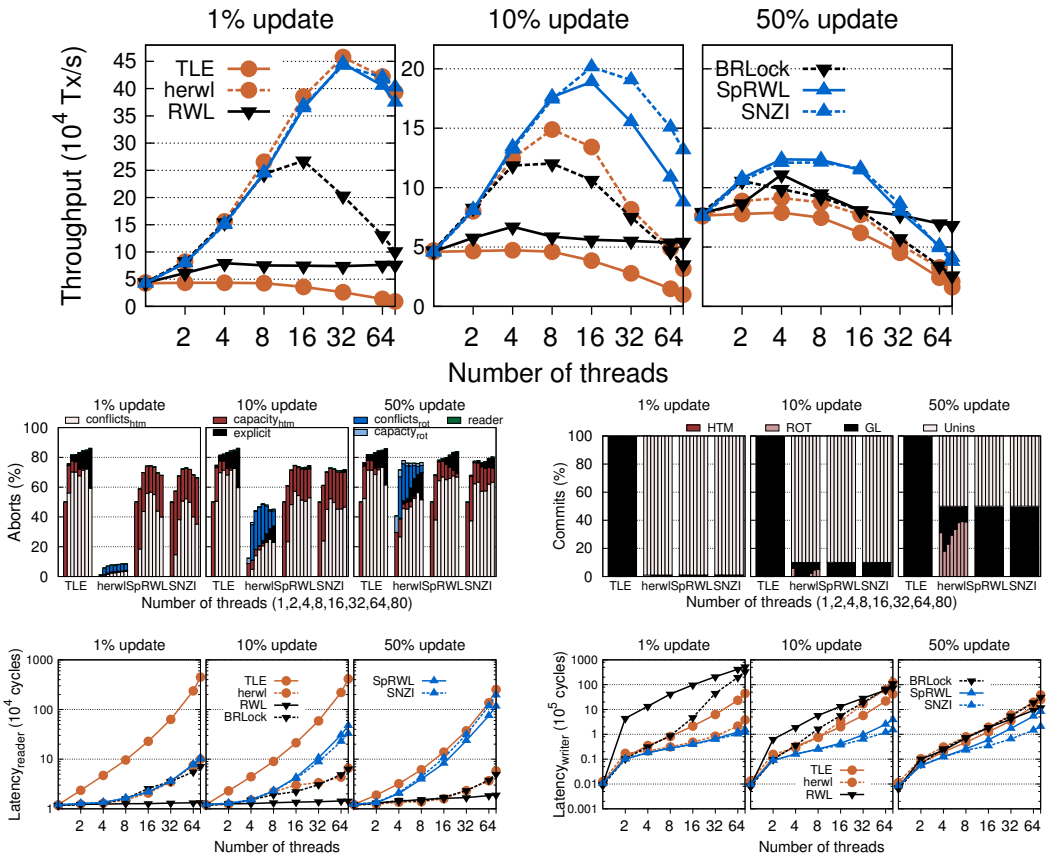

Figure 4.9: STMBench7: throughput, abort rate, and breakdown of commit modes at 1%, 10% and 50% update ratios on POWER8.

high thread counts, also on the Broadwell CPU, we observe a relevant (i.e., up to 7×) increase of the readers' latency — as already noticed on POWER8 with 1% of update operations. Again, this can be explained by the commit breakdown plot, which shows that only a small percentage of update transactions can actually be committed in HTM with this workload. However, glsSYS's gains in terms of writer latency's reduction latency are still sufficiently large to outweigh the increase of readers' latency (still, in comparison, to BRLock).

POWER8 shows trends similar to Broadwell at 10% and 50% of update operations. One noteworthy difference, though, is related to the use of SNZI for tracking readers, which had negligible, or even adverse, impact on throughput on Broadwell,

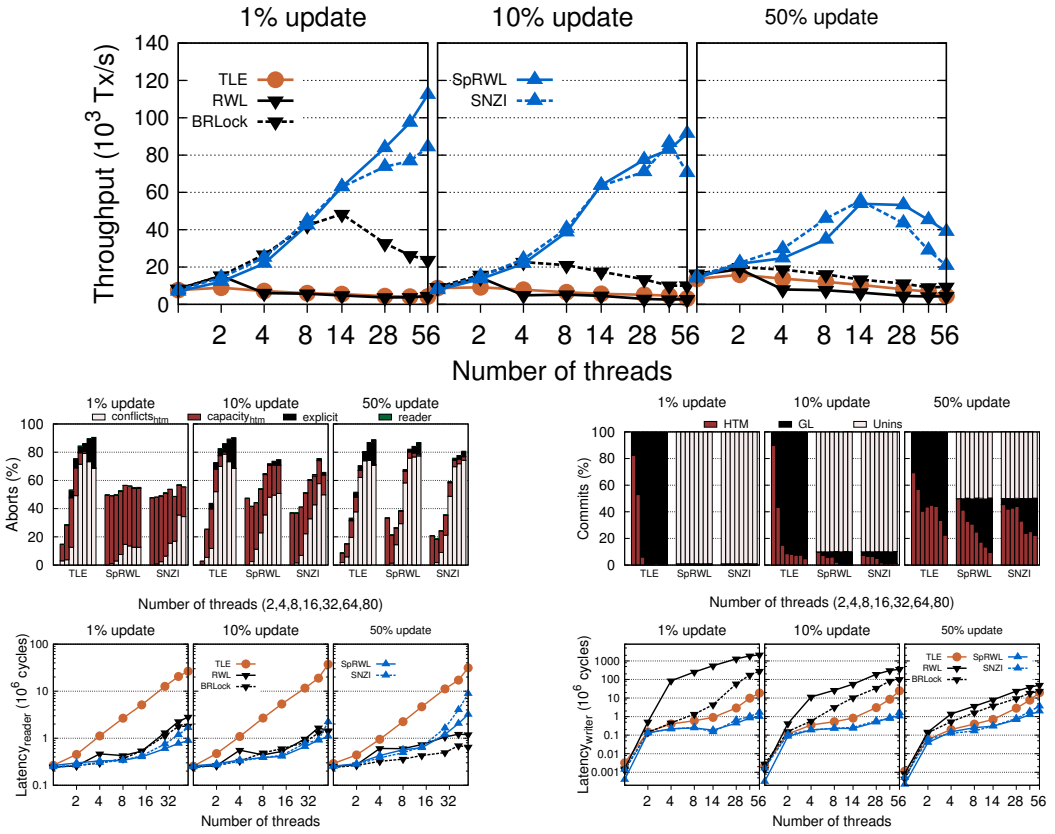

Figure 4.10: TPC-C. 1%, 10%, and 50% update operations. Stock-level (read-only) and Payment (update) transaction profiles on Intel.

brings instead relevant performance gains on POWER8, at least for the 10% update workload. As for the comparison with HRWLE, glsSYS achieves on par performance with 1% of update operations, but outperforms it by up to $\tilde{1}.5\times$ in both the 10% and 50% update operations' workloads. The reason of these gains is, also in this case, rooted back to the large overheads caused by the quiescence phase HRWLE requires ROTs to perform, in workloads, like this one, characterized by very long readers.

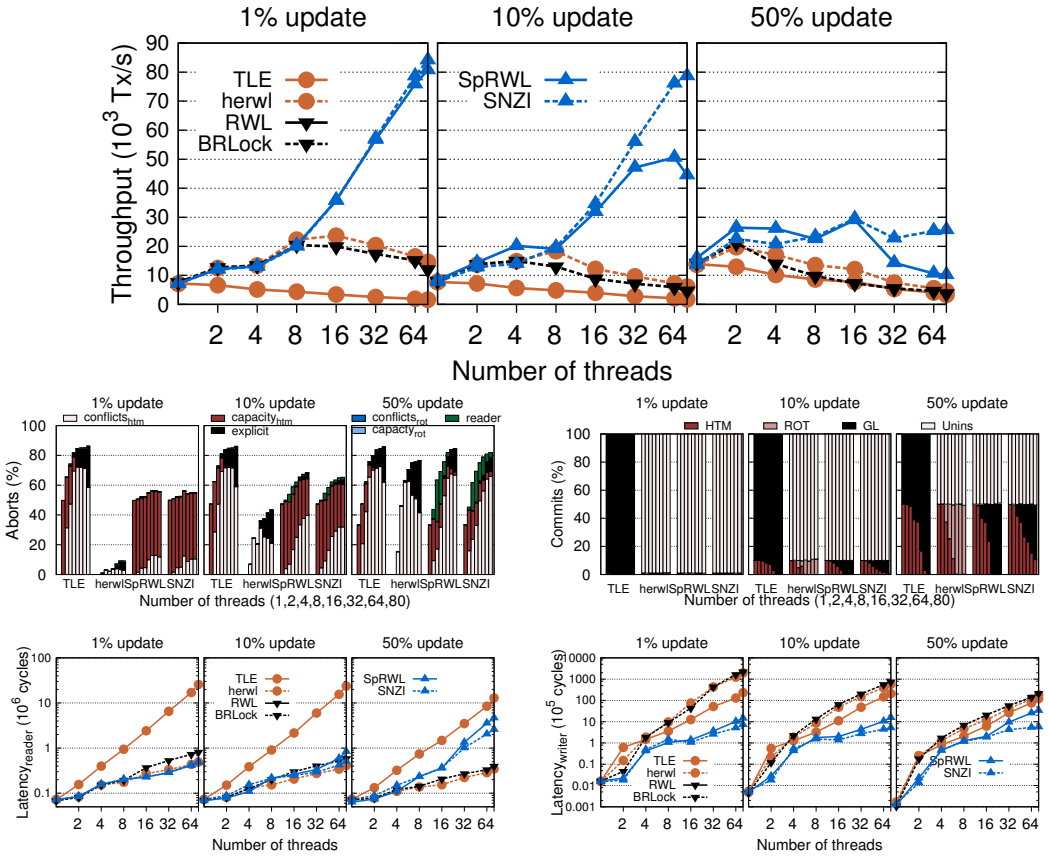

Figure 4.11: TPC-C. 1%, 10%, and 50% update operations. Stock-level (read-only) and Payment (update) transaction profiles on POWER8.

4.3 TPC-C

TPC-C is a well-known OLTP benchmark that simulates the workloads generated by a warehouse supplier application. TPC-C uses five different types of transactions, with very diverse profiles, such as long read-only transactions, long and contention-prone vs short and almost contention-free update transactions. We use this benchmark in two different tests: (i) a simple workload mix, which we denote as TPCC-1, composed by only two transactions profiles, Stock-Level, a long read-only transaction, and Payment, a short update transaction; (ii) a more complex transactional mix, noted TPCC-2, encompassing the whole set of transaction profiles defined by the benchmark in the following proportions: Stock-level, 31%, Delivery,

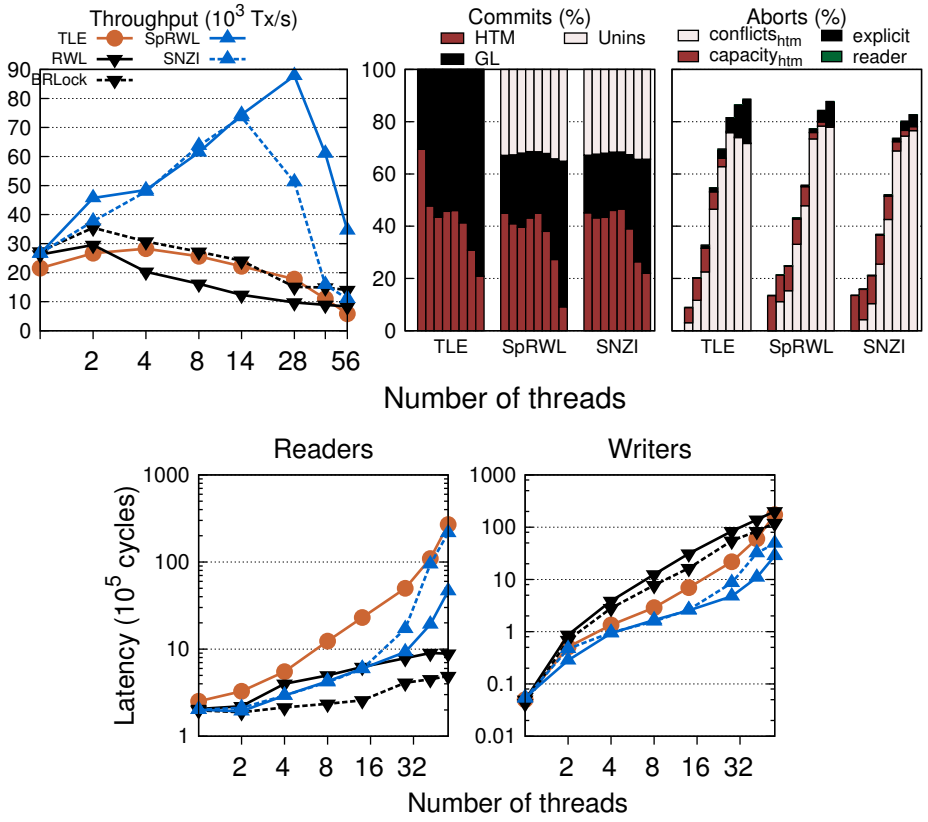

Figure 4.12: TPC-C. Mix comprising the following transaction profiles: Stock-level, 31%, Delivery, 4%, Order Status, 4%, Payment, 43%, and New Order, 18% on Intel

4%, Order Status, 4%, Payment, 43%, and New Order, 18%.

For the TPCC-1 workload mix, whose results we plot in Figure 4.10 and 4.11, we also vary the ratio of update operations in the set {1%, 10% and 50%}. With this workload mix, the scenario where SpRWLock achieves the largest gains is 10% updates, where it can generate throughputs up to 14×/15× times larger than BRLock in Broadwell/POWER8, respectively. In these settings, both SpRWLock variants achieve better or equal reader latency values as the most competitive baselines (BRLock on Broadwell and HRWLE on POWER8), while reducing writer latency by over 500×. Results also show that SpRWLock can scale to a much higher thread count than other backends, scaling up to 80 threads in u1%, in POWER8, both with SNZI and State Array variants.

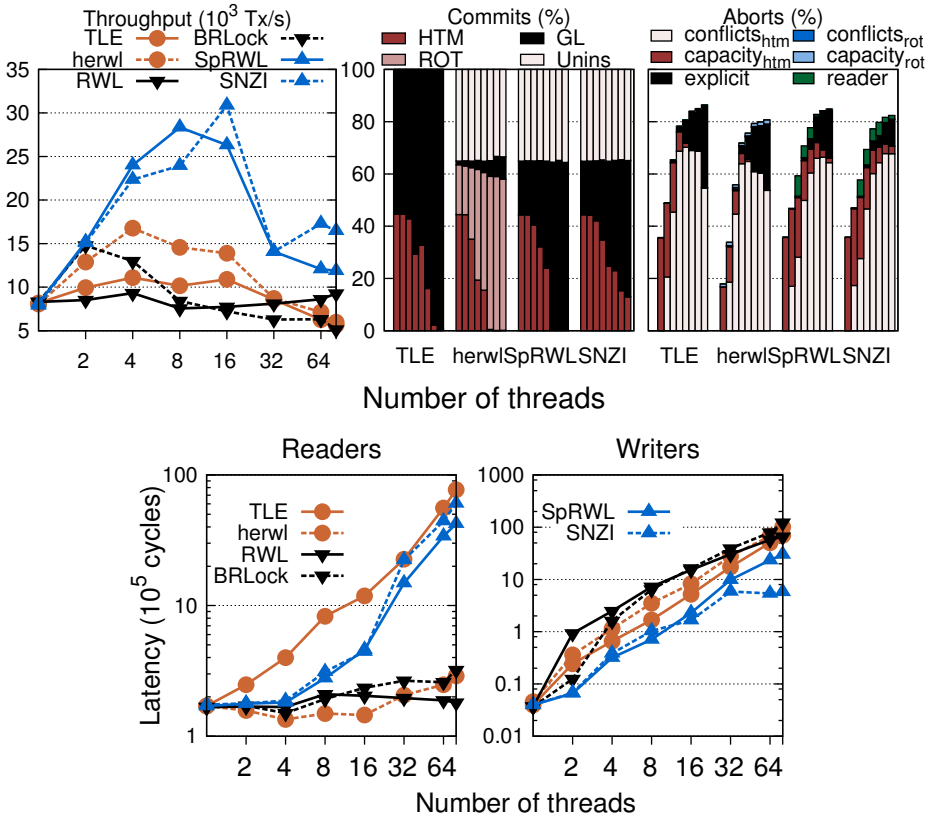

Figure 4.13: TPC-C. Mix comprising the following transaction profiles: Stock-level, 31%, Delivery, 4%, Order Status, 4%, Payment, 43%, and New Order, 18% on Power8.

Similar trends are observed also at 10% of updates, where we see, again on POWER8, that SNZI can play an important role in optimizing SpRWLock's performance. At 50% update operations, the workload scalability is significantly reduced, as, especially at high thread counts, the fraction of update operations that activate the fallback path increases significantly in this high contention workload. Yet, SpRWLock can still deliverable significant relative gains with respect to all the other baselines on both architectures.

Even when considering the much more complex TPCC-2 workload mix, see Fig.4.12 and 4.13, SpRWLock achieves remarkable throughput gains — up to 4.5×/2× on Broadwell and POWER8, respectively — and shows similar trends.

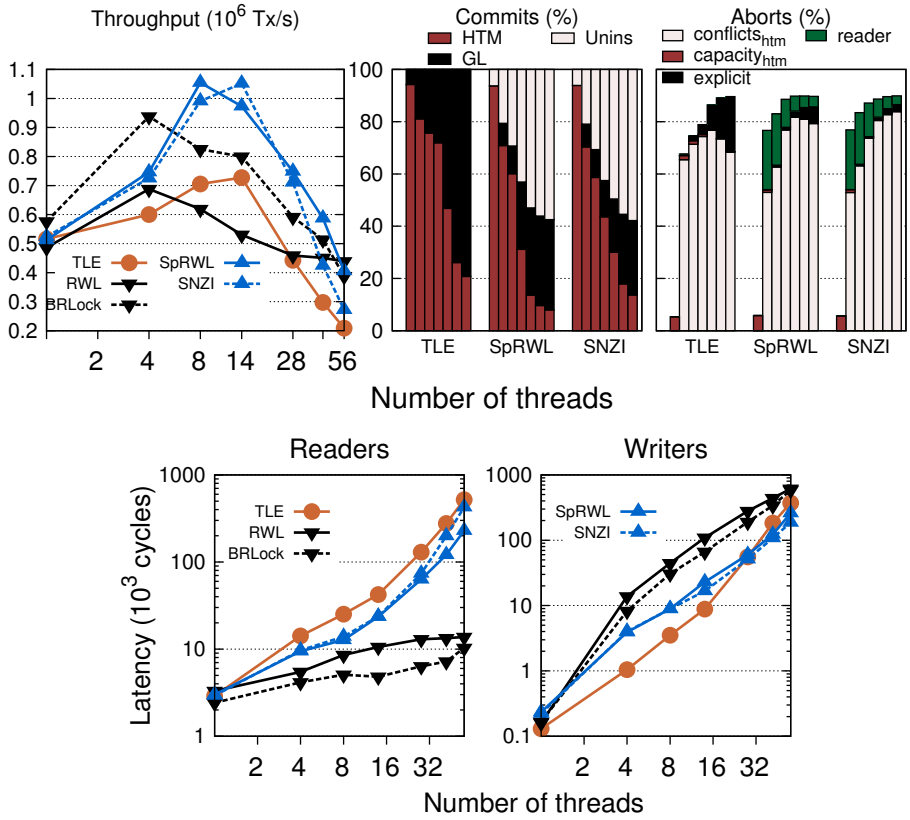

Figure 4.14: Kyoto: throughput, abort rate, and breakdown of commit modes for the wicked benchmark on Intel.

4.4 Kyoto Cabinet

We consider Kyoto Cabine [50] on its 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.

In order to work as close to our ideal workload as possible, Kyoto was tweaked to run its Wicked tests with bulk operations only. Overall, many of Kyotos read transactions seem to fit in HTM, avoiding the need for snzi or the state array, which reduces the difference in throughput seen between these two variants, it also allows us to see how our algorithm works in a realistic database management simulation.

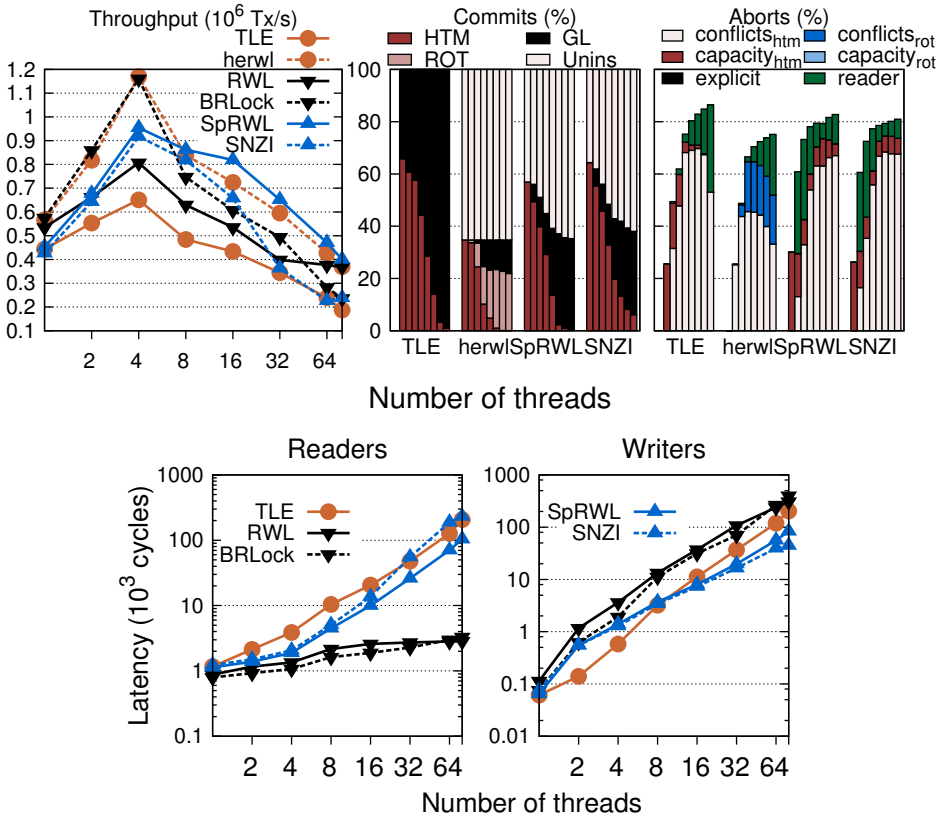

Figure 4.15: Kyoto: throughput, abort rate, and breakdown of commit modes for the wicked benchmark on Power8.

Throughput wise, SpRWLock achieves values up to 30% higher than typical backends, both on Intel and P8. As seen in previous benchmarks, SpRWLock continues to benefit writers, as their latency is 60%/30%, Intel and POWER8 respectively, lower than other backends (Figs. 4.14 and 4.15). In terms of scaling, SpRWLock scales up to 14 threads on Intel and 4 threads on POWER8, maintaining however a better throughput than other backends as threads count increases.

SpRWLock out-scales BRLock as thread count increases and although it does not achieve as good results as in previous benchmarks, having a lower throughput than BRLock at lower thread count, it still manages to outperform typical backends at higher thread count. As for latency, SpRWLock continues to show a higher reader latency throughout the whole plot, 20×/3.5× in Intel and POWER8 respectively,

and lower writer latency, achieving values up to 2× lower on Intel.

Chapter 5

Conclusions and Future Work

This thesis has designed, implemented and evaluated a novel synchronization mechanism, called SpRWLock (Speculative Read Write Lock), which allows to overcome, at least partially, a key limitation of current HTM implementations: its inability to accommodate the execution of transactions whose working set exceeds the capacity of the processor's cache.

SpRWLock is the first solution that allows for executing update transactions in HTM, while supporting the concurrent execution of uninstrumented read-only transactions (and hence spared from the capacity limitations of HTM), without requesting any special architectural feature from the underlying HTM implementation, i.e., by assuming a generic API for transaction demarcation.

SpRWLock was evaluated via an extensive experimental study conducted using the HTM implementations available on Intel's Broadwell and IBM's Power8 CPUs and encompassing synthetic micro-benchmarks aimed at assessing the sensitivity of the proposed solution to a broad spectrum of workloads, as well as standard benchmarks (TPC-C, STMBench7) and real applications (KyotoDB). The results of our study shows SpRWLock can yield throughput gains of up 15× with respect to both plain HTM -based solutions, as well as state of the art, non-speculative read-write lock implementations.

5.1 Future Work

During this work, a number of interesting ideas were identified, which can pave the way for future research:

- Dynamic Wait Value (Alpha): Before settling for the current SpRWLock implementation, when deciding how to implement the reader synchronization scheme, one of the options considered was to back-off readers for some predefined value, α. This value was initially set after offline testing, and, by carefully optimizing the value of α, we observer that it was possible to achieve perfomance similar to SpRWLock. The next step suggested would be to implement a sellf-tuning mechanism, e.g., based on gradient descent, aimed at identifying the optimal, workload-dependant, value of α.

- Predicting Reader in HTM (PRHTM): Another possibility enabled by the online statistics currently gathered by SpRWLock would be to implement a self-tuning scheme, which would automatically disable the use of hardware transactions for long readers that are likely to exceed the HTM's capacity limitations and execute them directly as uninstrumented readers.

- Reader Wait N: Another approach we have thought of is to allow small readers to start without waiting for active writers, if readers estimate to be able tocommit before writers do. This might, theoretically, increase throughput by reducing the average readers' wait. However, since we first attempt readers in HTM, readers that are small enough to finish before writers typically already fit in HTM. Nonetheless it is an interesting concept to explore outside HTM systems.

Bibliography

[1] 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.

[2] 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.

[3] E. Gaona, R. Titos, J. FernÃąndez, and M. E. Acacio. On the design of energy-efficient hardware transactional memoryâĂLsystems. *Concurrency and Computation: Practice and Experience*, 25(6):862–880, 2013.

[4] 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.

[5] 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.

[6] 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.

[7] z/Architecture Principles of Operation. SA22-7832-09.

[8] Nuno Diegues and Paolo Romano. Self-tuning intel restricted transactional memory. *Parallel Comput.*, 50(C):25–52, December 2015.

[9] 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.

[10] Colin Blundell, Christopher Lewis, and Milo M K Martin. Deconstructing transactional semantics: The subtleties of atomicity. 06 2005.

[11] Yehuda Afek, Amir Levy, and Adam Morrison. Programming with hardware lock elision. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 295–296, New York, NY, USA, 2013. ACM.

[12] 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.

[13] TPC Council. TPC-C Benchmark, 2011.

[14] 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.

[15] FAL Labs. Kyoto cabinet: A straightforward implementation of DBM, 2011. http://fallabs.com/kyotocabinet/.

[16] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking concurrency. *Queue*, 4(10):24–33, December 2006.

[17] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Commun. ACM*, 14(10):667–668, October 1971.

[18] Jonathan Corbet. Big reader locks. https://lwn.net/Articles/378911.

[19] 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.

[20] Paul E. McKenney and John D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[21] 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.

[22] 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.

[23] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: Streamlining stm by abolishing ownership records. *SIGPLAN Not.*, 45(5):67–78, January 2010.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th An-*

nual ACM/IEEE International Symposium on Microarchitecture, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[31] 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.

[32] 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.

[33] 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.

[34] Moir Bussam, Lev. Phtm: Phased transactional memory. 2007.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] David Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive integration of hardware and software lock elision techniques, 06 2014.

[40] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[41] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. Snzi: scalable nonzero indicators, 08 2007.

[42] Jonathan Corbet. Big reader locks, 2016.

[43] 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.

[44] Diego Rughetti, Paolo Romano, Francesco Quaglia, and Bruno Ciciani. Automatic tuning of the parallelism degree in hardware transactional memory. In *Euro-Par 2014 Parallel Processing*, pages 475–486. Springer International Publishing, 2014.

[45] 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. In *Proceedings of the*

42Nd Annual International Symposium on Computer Architecture, ISCA '15, pages 144–157, New York, NY, USA, 2015. ACM.

[46] https://github.com/tsepol/SpRWL, 2018.

[47] 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.

[48] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[49] E. Jones. tpccbench. https://github.com/evanj/tpccbench, 2017.

[50] FAL Labs. Kyoto cabinet: A straightforward implementation of DBM, 2011. http://fallabs.com/kyotocabinet/.