Hardware Approaches to Transactional Memory in Chip Multiprocessors

J. Rubén Titos-Gil and Manuel E. Acacio

1 Introduction

Multicores are nowadays at the heart of almost every computational system, from the smartphone in our pocket, to the server-class machines in datacenters that provide us with a myriad of cloud services. With the advent of chip multiprocessors, the shift to mainstream parallel architectures is inevitable, and both programmers and architects are presented with immense opportunities and enormous challenges. Despite the fact that multiprocessor systems have existed for a long time, multi-threaded programming has not been much of a focus. Instead, multiprocessors were of interest only to the small community of high-performance computing (HPC), and so was parallel programming, which was mostly ignored by software vendors, and not widely investigated nor taught. As a matter of fact, most software development over time has been predicated on single-core hardware, and the collective knowledge of software developers across organizations has been based primarily on single processor platforms.

Now that the *free lunch* is over [76], programmers must change the way they create applications to fully leverage multicore hardware. At every layer of the computing stack, whether the targeted platform is a handheld device or a warehouse-scale computer, programmers are being pushed towards unfamiliar programming models in order to deliver parallel software that takes advantage of the newly available computational resources and meets the demands of the end user. In the context of datacenters, the task is even more daunting because of the massive scale and complex architecture of these systems where efficient exploitation of parallelism is paramount at every level. Ideally, parallel software developed for these large-scale clusters should be able

J. R. Titos-Gil (✉)
Chalmers University of Technology, Gothenburg, Sweden
e-mail: ruben.titos@chalmers.se

M. E. Acacio
Universidad de Murcia, Murcia, Spain
e-mail: meacacio@ditec.um.es

S. U. Khan, A. Y. Zomaya (eds.), *Handbook on Data Centers,*
DOI 10.1007/978-1-4939-2092-1_27

to harness the potential of their multicore building blocks, while improving aspects that impact the total cost of ownership such as energy efficiency, server utilization, code maintainability or programmer productivity. Hybrid programming models that use shared memory for intra-node parallelism and message passing for inter-node communication are a good example of how programmers exploit these large-scale systems with multi-core processors. New programming models keep appearing in today's datacenters as a result of the wide spectrum of applications and their diverse characteristics. On the one hand, traditional HPC datacenters usually run scientific workloads that have long, computationally-intensive jobs, often as a single binary exclusively executed on a large number of nodes, where synchronization and communication abounds. On the other hand, Internet services exhibit ample parallelism given their large data sets of relatively independent records (e.g. web pages) and the thousands of independent requests received per second. In either environment, the programmer's job is to find the most appropriate way to efficiently exploit the parallelism that is inherent to the problem, maintaining high productivity while producing correct code that is easily verifiable and composable.

Many applications that run in today's datacenters have very strong requirements in terms of response time. This is particularly true for those online services that provide an almost instantaneous reply to the user, such as a web search engine. While the work required to process a user's request can be rather easily partitioned across different nodes in independent units of data, each task that executes in a single node generally performs a substantial amount of computation due to very large data sets. This alleviates the overheads inherently imposed by the communication and synchronization of hundreds or thousands of parallel tasks amongst different nodes of the datacenter. Given the considerable extent of the job performed by each task, the algorithms executed at the task level may also be subject to parallelization in order to speedup the task and reduce overall latency. This could improve utilization of the datacenter too, addressing the important trade-off between keeping machines busy and response times low. Since each task is mapped to a single computing node where all processing cores share the same address space, the intuitive abstraction of shared memory may simplify programmer's job of turning a monolithic task into a parallel, multi-threaded program.

A fundamental problem that all programmers face when writing multi-threaded code is the difficulty of simultaneously achieving both high efficiency and productivity/correctness. Designing a parallel algorithm involves orchestrating the concurrent execution of the parts to improve performance while at the same time guaranteeing correctness. Complex and hard-to-find, software defects unique to multi-threaded applications such as race conditions and deadlocks can quickly derail a software project [18]. Software engineering tools have yet to simplify the programming for these shared-memory architectures in order to make the new hardware resources accessible to the average programmer. In order to avert a software crisis, developers must adapt and improve such tools to make them better suited for parallel multicore software development [83]. The reality is that software has not matured enough to take advantage of the number of cores that are already available in today's systems, and the vast majority of applications are still single-threaded [30]. The rise

of multicores has brought such problem of effective concurrent programming to the forefront of computing research. To help alleviating this problem, *Transactional Memory* (TM) has been proposed as a concurrency control mechanism that aims to simplify concurrent programming with reasonable scalability.

This chapter examines the state-of-the-art of Transactional Memory, paying special attention to its hardware implementations (*Hardware Transactional Memory* or HTM). Recent inclusion of HTM support in commodity multicore processors (Intel's Transactional Synchronization Extensions [91]) and commercial mainframes (IBM's Transaction Execution Facility [40]) has converted TM into a reality for current and future datacenters.

The remainder of this chapter is organized as follows. In Sect. 2 we delve into the problems that traditional parallel programming with locks has and discuss how TM can alleviate them. Subsequently, we present the fundamentals of TM in Sect. 3 and the hardware mechanisms TM requires in Sect. 4. Next, we describe the programming interfaces of the hardware TM support provided by the new Intel processors (Sect. 5), and present a brief performance analysis of it (Sect. 6). Section 7 summarizes the most relevant proposals found in the HTM research literature. The main conclusions of this chapter are summarized in Sect. 8.

2 Why Transactional Memory Is Going Mainstream

Concurrent programming is a far more challenging task than sequential programming: A parallel program is undoubtedly more difficult to design, write, and debug than its sequential counterpart. Orchestrating the concurrent execution of the parts to improve performance while at the same time guaranteeing correctness is by no means an easy task. Designing parallel algorithms requires restructuring code and data in often counter-intuitive ways, so that it can be split into parallel tasks. Balancing the workload among the available processors, or communicating and managing shared data between different processors are some of the many factors that make parallel programming a complicated endeavour. Programmers need to reason carefully about possible interactions of their threads when running concurrently, and not doing so may result in programs that are incorrect, perform poorly, or both. To add insult to injury, parallel programs are very hard to debug due to the combinatorial explosion of possible execution orderings: Parallel programs often produce non-deterministic results, making it harder to prove programs correct, and their bugs are often elusive and notoriously difficult to find and fix, because of the difficulty to reproduce the exact same execution (i.e. interleaving of threads, etc.) that leads to a race.

In the context of shared memory architectures where concurrent tasks process shared data, guaranteeing correctness while maintaining efficiency and productivity is a key challenge. Parallel thread execution requires synchronization for accessing shared data. Programmers are responsible for ensuring that concurrent accesses to shared data structures are correct, and often rely on mutual exclusion mechanisms to protect these critical sections, so that no more than one thread can simultaneously enter the same critical section and access the same shared data.

2.1 The Drawbacks of Lock-Based Synchronization

Traditional multi-threaded programming models use low-level primitives such as locks to guarantee mutual exclusion. Unfortunately, the complexity of lock-based synchronization makes parallel programming an error prone task, particularly when fine-grained locks are used to extract more parallelism. At one end, heroic programmers seeking performance try to minimize the amount of shared resources (data) that are protected by the same lock, so that different threads accessing different data do not have to serialize their execution unnecessarily, thus enabling maximum concurrency. However, the use of fine-grain locks adds more programming complexity, since programmers must be careful to acquire them in a fixed, predetermined order so as to avoid deadlocks. At the other end, common programmers seeking productivity (correctness) choose to reduce the complexity of reasoning, i.e. likelihood of deadlock, by using fewer locks with coarser granularity where each lock is responsible for protecting larger critical section. This naturally comes at the cost of sacrificing performance, when threads without true data races contend for the same lock. Though programmers can also include deadlock detection mechanisms in their programs, to try and recover from deadlocks, this alternative also adds substantial complexity.

As if deadlocks were not enough, locking brings about other undesired situations like priority inversion (when a high priority thread is unable to acquire a lock because a lower priority thread is holding it), convoying (when a lock holder is de-scheduled from execution, impeding others to progress) and lack of fault tolerance (when a lock holder modifies data and then crashes, causing the whole program to fail). Furthermore, locking breaks the abstraction principle, as programmers using a module need to be aware of the locks it uses, to ensure that the program still follows the predetermined locking order that prevents deadlock. Therefore, locks jeopardize the code composability property, as two individually correct modules can deadlock when combined together.

2.2 The Transactional Abstraction

The trade-off between programming ease and performance imposed by locks remains one of the key challenges to programmers and computer architects of the multicore era. Transactional Memory (TM) [34, 36] has been proposed as a conceptually simpler programming model that can help boost developer productivity by eliminating the complex task of reasoning about the intricacies of safe fine-grained locking. TM inherits the concept of *transaction* from the database community, and applies it to the domain of shared-memory programming in an attempt to simplify the task of thread synchronization. Transactions in the multi-threaded programming world are blocks of code that are guaranteed to be executed atomically and in isolation with respect to all other code. At a high level, the programmer or compiler annotates sections of the code as atomic blocks or transactions. The underlying system then executes these transactions speculatively in an attempt to exploit as much concurrency as possible.

TM systems generally employ an optimistic approach to concurrency control in order to let multiple transactions execute in parallel, while still preserving the properties of atomicity and isolation. Therefore, the TM system attempts to make best use of available concurrency in the application while guaranteeing correctness. By using transactions to safely access shared data, programmers need not reason about the safety of interleavings or the possibility of deadlocks to write correct multi-threaded code. Hence, TM addresses the performance-productivity trade-off by not discouraging programmers from using coarse-grain synchronization, since the underlying system can potentially achieve performance comparable to fine-grained locks by executing transactions speculatively. In addition to addressing such critical trade-off, TM tries to solve other limitations of lock-based synchronization. Transactional code is robust in the face of both hardware and software failures, as the system can always rollback the speculative updates to its pre-transactional state in case a thread crashes inside a transaction. Unlike locks, transactions are composable, and they can be safely nested without any risk of deadlocks [6].

2.3 High-Performance Transactional Memory

Transactions are a promising abstraction that could ease parallel programming and make it more accessible to the common programmer. Transactional semantics can be entirely supported in software, hardware, or using a combination of both. According to this, we can classify TM systems into software transactional memory (STM), hardware transactional memory (HTM), and hybrid transactional memory systems.

STM implementations [26, 35, 48, 72] allow running transactional workloads on existing systems without requiring special hardware support, providing a great degree of flexibility at little cost. Unfortunately, implementing the necessary mechanisms entirely in software imposes too high an overhead and thus STM systems do not fare well against traditional lock-based approaches when performance is important. For this new paradigm to be a viable alternative to locks, the key mechanisms that provide transactional semantics must be implemented at the architectural level.

Hybrid TM systems [4, 12, 23, 43, 74, 77] attempt to combine both the speed and flexibility by using simple hardware to accelerate performance-critical operations of an STM implementation. In this way, hybrid implementations of TM rely on some kind of software intervention to execute transactions, though they minimize the overheads of providing transactional semantics in comparison to a software-only solution. Hybrid TM models use the STM as a backup to handle situations where the hardware cannot execute the transaction successfully [34].

Transactional semantics can also be supported largely in hardware [1, 14, 16, 31, 50, 51, 89], allowing for good performance with varying degrees of complexity, which change considerably from one HTM proposal to another depending on what kind of transactions the TM system is capable of committing without resorting to fallback mechanisms. Simple HTM schemes [17, 22, 36] adopt a "best-effort" solution that cannot not guarantee that all transactions will eventually commit successfully

using hardware support alone, mostly because of the limitations imposed by the hardware structures involved. More sophisticated HTM proposals [1, 31, 51] address this limitation in transaction size, guaranteeing that certain "bounded" transactions can be entirely executed in hardware. These proposals typically behave in the same way as best-effort ones as long as hardware structures are sufficient, and then fall back to additional hardware mechanisms to maintain transactional properties on resource overflow. However, neither bounded nor best-effort solutions can commit transactions that encounter events that are too complicated to handle in hardware, like context switches, page faults, I/O, exceptions or interrupts [37], and in such circumstances the transaction is invariably aborted. Even more elaborated HTM schemes have been designed [1, 64] to handle all transactions in hardware, ensuring that the same transaction is not indefinitely aborted because of its size, duration or other events it may encounter. Unfortunately, the complexity of these "unbounded" HTM designs makes them too costly for processor manufacturers to consider them in practice.

2.4 Industrial Adoption of Hardware Transactional Memory

In the early 2000s, Transmeta was the first company to implement a form of transactional memory in its x86-compatible Crusoe microprocessor, though this hardware only meant to support aggressive speculative optimizations in its dynamic binary translation system [24].

More recently, Azul Systems included HTM support in its Vega systems [22], a specialized appliance designed to massively scale the usable compute resources available to Java applications. However, the HTM support was only used to accelerate Java locks and not exposed to programmers.

Sun Microsystems was the first general-purpose processor manufacturer that ventured to introduce support for transactions in a chip multiprocessor. In 2007, the company announced that its high-end Rock processor would have support for both transactional memory and speculative multithreading [17]. Unfortunately, Sun cancelled the project in 2009, and Rock chips never made it to the market, though some prototypes were distributed for research purposes.

Around the time the Rock project was cancelled, AMD proposed the Advanced Synchronization Facility (ASF) [21], a set of instruction extensions to the x86 architecture that provide limited support for lock-free data structures and transactional memory. To date, it is unknown whether any future AMD products will implement ASF.

In mid 2011, IBM revealed that its BlueGene/Q compute chip would feature transactional memory support. The custom design was a system-on-a-chip that integrated 18 PowerPC cores with memory and networking subsystems [33]. Cores shared a multiversioned L2 cache which supports transactional memory and speculative multithreading. With the lessons learned from BlueGene/Q, IBM began to ship the IBM zEnterprise EC12 system in the fall of 2012, less than a year after the first BlueGene/Q system made its debut in the Top500 list. The zEC12 processor introduces

the Transactional Execution Facility [40], which extends the z/Architecture used on IBM mainframes with transactional memory support. The zEC12 has given IBM the distinction of becoming the first company to deliver commercial chips with this technology [27].

In early 2012, Intel announced that its new *Haswell* microarchitecture would implement hardware transactional memory through a set of new instructions called *Transactional Synchronization Extensions* (TSX) [61, 91]. Shortly after, Intel's TSX specification was released, describing how TM is exposed to programmers, but withholding details on the actual implementation. In mid-2013, Intel began shipping processors based on its 4th-generation Core microarchitecture, making the Core i3/i5/i7 and Xeon v3 processor families the first chips with TM support that are available in the consumer and server markets. The adoption of transactional memory by mainstream, commodity x86 processors culminates a two decade journey of active academic research. Section 7 provides a good overview of the contributions have brought the industry here.

3 Fundamentals of Transactional Memory

Transactional Memory (TM) [34, 36] has been proposed as an easier-to-use programming model that can help developers build scalable shared-memory data structures, relieving them from the burdens imposed by fine-grained locking. Under the TM model, the programmer declares *what* regions of the code must appear to execute atomically and in isolation (called *transactions*), leaving the burden of *how* to provide such properties to the underlying levels. The TM system then executes optimistically transactions, stalling or aborting them whenever real run-time data races (called *conflicts*) occur amongst concurrent transactions. The TM programming model thus replaces explicit synchronization mechanisms like locking with a more declarative approach whose aim is to decouple performance pursuit from programming productivity. The transactional abstraction is provided at the programming language level through a new construct, e.g. `atomic`, employed by programmers to delimit accesses to shared data—i.e. *critical sections*—thus structuring their parallel code into *atomic blocks* or transactions. A transaction is said to *commit* when it completes its execution successfully—confirming its speculative updates to shared memory—while it is *aborted* or *squashed* when some condition occurs—e.g. a conflict with a concurrent transaction—that impedes its completion with success. To guarantee race-free execution of a transactional multi-threaded application, TM implementations must satisfy two basic properties, namely atomicity and isolation, which are inherited from the database domain.

The *atomicity* property dictates that a transaction is either executed to completion or not executed at all. If the transaction successfully commits, all of its speculative changes are made globally visible at once. Otherwise, if the transaction aborts, all its tentative updates are discarded in order to revert the system to its pre-transactional state, as if the transaction had never executed. To the outside world, this means that

a transaction appears as an indivisible operation that cannot be partially executed. On its part, the *isolation* property requires that the intermediate (speculative) state of a partially completed transaction must remain hidden from other code. By satisfying these properties, transactions appear to execute in some serial global order, i.e. committed transactions are never observed by different processors as executed in different orders. To provide these properties, the TM system must implement two basic mechanisms, namely data versioning and conflict management. The policy and implementation of these two mechanisms constitutes the two fundamental dimensions of the TM design space.

Version management handles the simultaneous storage of both speculative data (new values that will become visible if the transaction commits) and pre-transactional data (old values retained if the transaction aborts). Only one of the two values can be stored *in-situ*, i.e. in the corresponding memory address, while the other needs to be placed somewhere else. The data versioning policy dictates how the system handles the storage of both versions, and it constitutes a major design point of the system. Depending on which value, old or new, gets to stay "in place" during the course of the transaction, the data version management policy can be classified as eager or lazy. Lazy versioning keeps old values *in-situ* until the commit phase, buffering speculative updates "on the side" in the meantime. In contrast, an eager approach to versioning uses a per-thread *transaction log* to backup the old value of a memory location prior to each write, and then updates the memory location with the new value.

When two concurrent transactions access the same memory location, and at least one of the accesses is a write operation, we say that there is a *conflict* or *race* between them. TM systems implement a *conflict management* mechanism to detect and resolve such conflicts. For this purpose, the data read and written by each transaction must be tracked. The set of data addresses that a transaction modifies during its execution is known as *write set*. Similarly, the *read set* refers to the group of memory locations read by the transaction. In these terms, a conflict between two concurrent transactions happens when a transaction's write set overlaps with other concurrent transactions' read or write sets. Depending on the meta-data information used for transactional book-keeping, conflict detection can take place at different levels of granularity, from objects, to cache lines to word or even byte-level addresses.

4 Hardware Mechanisms for Transactional Memory

HTM systems must identify memory locations for transactional accesses, manage the read-sets and write-sets of the transactions, detect and resolve data conflicts, manage architectural register state, and commit or abort transactions [34].

4.1 ISA Extensions

Identifying transactional boundaries is accomplished by extending the instruction set architecture (ISA). All HTM implementations introduce a pair of new instructions,

i.e. "begin transaction" and "commit transaction", to delimit the scope of a transaction. On the one hand, the execution of the "begin transaction" instruction causes the processor to enter into "transactional mode" (usually setting some bit in the status register) and perform some common actions related to the initialization of the basic transactional mechanisms, like checkpointing the architectural registers to a shadow register file. The architectural registers and memory combined form the precise state of the processor, and therefore the register state also needs to be restored to a known precise state in case of abort. The operation of creating a shadow copy of the architectural registers at the start of a transaction is rather straightforward and can often be performed in a single cycle. On the other hand, the "commit transaction" instruction attempts to confirm the speculative updates of the transaction by publishing them to the rest of the system, and it returns the processor to non-transactional state if successful, discarding the register checkpoint.

The most straightforward step to identify transactional accesses is to leverage these two instructions that mark the beginning and end of a transaction, so that all the loads and store instructions executed while in transactional mode are implicitly considered transactional. This is the approach that most modern HTM proposals follow, including the Haswell microarchitecture [91], and the failed Rock processor [17]. Another option is to further augment the ISA with explicit "transactional load" and "transactional store" instructions, separated from their conventional counterparts. Though allowing a transaction to contain both transactional and non-transactional accesses may complicate things, this provides increased flexibility and may aid programmers to reduce the pressure on the underlying TM mechanisms, as non-transactional accesses do not participate in data versioning nor conflict detection. The original HTM proposal by Herlihy and Moss [36] as well as the AMD Advanced Synchronization Facility (ASF) [21] are explicitly transactional designs.

Some proposed HTMs also include an "abort transaction" instruction to explicitly roll back the tentative work of transaction. This is an example of flexible design that may enable TM hardware to be applied toward solving problems beyond guaranteeing mutual exclusion during the execution of critical regions. Programmers using hardware transactions may find useful the ability to explicitly rollback execution upon a certain condition, which need not necessarily be a conflict with other transaction.

4.2 Transactional Book-Keeping

HTM systems must track a transaction's read and write set in order to detect data races amongst concurrent transactions. Many HTMs extend the cache line metadata kept at the private cache level, with two new bits that record, respectively, whether the line has been speculatively read (SR) and/or speculatively modified (SM) during the ongoing transaction [31, 51, 82]. Such designs also support the capability to clear all the read bits in the data cache instantaneously, an action that is performed when the transaction commits or aborts. The private caches serve as a natural place to track a transaction's read and write sets, enabling low overhead tracking, although they also constrain the granularity of conflict detection to that of a cache line.

All HTM systems that leverage the private level cache to perform transactional book-keeping are susceptible of transactional *overflows* due to the cache's limited capacity or associativity. Best effort designs would automatically abort the transaction if a cache line whose SR or SM bit is set is replaced, while bounded schemes would resort to some safety net in order to keep tracking read and write sets and detecting conflicts in the presence of spilled lines.

An alternative scheme of transactional book-keeping which does not leverage the private level cache is to use Bloom filters to conservatively summarize a transaction's data accesses using "address signatures" [14, 89]. The main disadvantage of hash encoding is that false positives may signal spurious conflicts, this is, the signature may indicate that an address belongs to the transaction read and write sets when in fact it does not.

4.3 Data Versioning

Besides keeping read and write set metadata, private caches are the natural place to buffer speculative values, since they are on the access path for the local processor and thus can automatically forward the latest transactional update to subsequent loads without special search. Write-back caches can be modified to behave as write buffers that support versioning of speculative data. Depending on the implementation, one or multiple versions of a speculatively modified cache line may be allowed.

For HTM systems with eager version management [51, 89], caches need no changes as they do not really have any notion of speculative writes. All writes go to the memory hierarchy in the same way, whether they occur inside or outside a transaction. It is then the responsibility of the coherence protocol to detect accesses to speculatively written data, and ensure no other threads or transactions observe it. Unlike lazy systems, evictions of speculatively written data from the private caches are tolerated, and they need no special treatment from the point of view of the versioning mechanism. However, specialized hardware is required to fill this virtualized log with the old value of each memory location that is being updated inside a transaction. The contents of the log are simply discarded on commit, by resetting the log pointer to its initial position. On an abort, a software handler walks the log restoring the original values into memory.

4.4 Conflict Detection and Resolution

HTM proposals leverage coherence mechanisms for conflict detection. Invalidation-based, cache coherence protocols allow HTM implementations to detect conflicts among concurrently running transactions at the granularity of cache lines. While unnecessary transactional conflicts may arise as a result of false sharing, for most transactional workloads this choice of granularity represents a good trade-off between design cost and performance.

More important than the granularity of the detection is the design decision of *when* conflicts with concurrent transactions must be detected. Strategies for conflict detection and resolution vary depending on when a processor examines the bookkeeping information of its read and write sets. In systems with eager detection—sometimes also referred to as pessimistic—conflicts are detected as soon as they happen, i.e. on every individual memory access. In the opposite approach, called lazy or optimistic conflict detection, this check is delayed until transaction commit, and the resolution is generally a committer-wins scheme.

The coherence protocol already provides mechanisms to locate the copies of a requested cache line, and thus the detection of transactional conflicts can be achieved with straightforward extensions. In snooping-based protocols, all caches observe all coherence traffic for all lines, allowing cache controllers to check for conflicts whenever a request is observed on the bus. In directory-based protocols, cache controllers only observe the coherence traffic corresponding to the lines that are currently privately cached. In a typical MESI directory protocol, a local store to a shared (S) line results in a write miss, since the protocol ensures that no cache can have permissions to write the data at this point. An exclusive request is sent to the directory, which in turn forwards invalidation messages to the current sharers of the line (except maybe the requester). The sharers are then able to check whether the requested address belongs to their read set—by checking the SR bit in cache, the read signature, etc.—and appropriately detect a *write-read* conflict. Similarly, a load (store) miss to a line that is remotely cached in modified (M) or exclusive (E) state results in a read (write) request forwarded from the directory to the cache that has the latest copy of the data, which then checks its write-set (read- and write-set) metadata to determine if a *read-write* (*write-write*) conflict exists.

Once an HTM system detects a conflict, it must determine how to resolve it. The conflict resolution policy constitutes another design dimension in HTM by dictating which transaction wins the conflict and is granted access to the data. The loser transaction can stall its execution, or it can be aborted: The alternatives change depending on when the conflict is detected.

In HTMs with eager conflict detection, there are several policies for resolution: requester wins, requester aborts, or requester stalls using a scheme of conservative deadlock avoidance. The implementation of the requester wins policy is straightforward: The cache or caches that detect a conflict simply trigger abort and yield to the requester. If the conflicted data was not speculatively modified (write-read conflict), the loser responds with the appropriate invalidation acknowledgement or data message. Otherwise, the response may be delayed until the data is conveniently restored. The main drawback of this policy is that it can produce livelock scenarios. The opposite option is to abort the requester. This is accomplished by augmenting the coherence protocol with negative acknowledgements (*nack*) messages, so that a cache controller that detects a conflict responds to a forwarded request or invalidation with a *nack* message. On reception of a *nack* response, the requester knows it has lost the conflict and can take the appropriate actions. The simplest alternative is to trigger its own abort, but this can also result in livelock. A less draconian, livelock-free solution is to stall the transaction and periodically retry the conflicting memory

access until a positive response (different from the *nack*) is received. In this case, cyclic dependencies amongst transactions can bring the system to a deadlock, and so the system must have a way out such possible cycles. LogTM [51] uses a simple timestamp-based scheme to conservatively detect cycles, aborting the youngest transaction to break the possible cycle.

HTM systems with lazy conflict detection must resolve conflicts when a committer seeks to commit a transaction that conflicts with one or more other transactions. The resolution policy in this scenario can abort all others, or else stall or abort the committer. In general, lazy HTMs follow a committer wins policy [10, 32] that favours forward progress and is both deadlock- and livelock-free. Unfortunately, the committer wins policy does not guarantee fairness and can result in starvation for some transactions.

4.5 Transaction Commit

The execution of the "commit transaction" instruction attempts to make the transaction's tentative changes permanent and visible to other processors instantaneously. Such publication is in itself a task that must occur atomically and without interference from other processors. For most HTMs, publishing speculative updates means obtaining exclusive ownership for all cache lines in the write set, and then releasing isolation over both transactional sets at once.

The implementation of the commit instruction is a straightforward operation in eager HTMs, since writes were performed in place and therefore all write set lines held in cache have write (exclusive) permissions. As for lazy HTMs, the requirement of en-masse publication of speculative updates to shared memory at commit time poses more challenges when multiple speculative versions of the same data can coexist. Commits in this case are non-trivial because each SM line must be located and its coherence permission upgraded while every other copy in remote caches gets invalidated. On the other hand, lazily-versioned HTMs that allow at most one speculative version easily provide local commits since speculative writes are performed only when the protocol has obtained exclusive ownership (write permissions) for the line. To simultaneously support both local commits and aborts, the coherence protocol must be able to tolerate silent replacements of exclusively owned lines, and it must be adapted to ensure the consistent version of the data is always written back to the shared levels of the memory hierarchy before the first speculative write.

4.6 Transaction Abort

A hardware transaction may be implicitly aborted by the conflict resolution mechanism, or the abort can be explicitly triggered from the program via an "abort transaction" instruction. Aborting a transaction means discarding all its tentative

changes and return the state of the processor to the exact same state it was right before the transaction began. Book-keeping information (SM and SR bits, signatures, etc.) must always be cleared on abort, and the last step of the abort process is the restoration of the architectural registers using the checkpoint that was saved in the shadow register file at the beginning of the transaction.

Implementing the abort functionality is quite simple in lazy HTMs, since speculative writes were performed "on the side" (in private structures local to the core) and therefore the shared memory still contains consistent, pre-transactional values. Aborts are cheap since silent invalidations of shared-state lines are generally supported by the protocol, and thus lazy HTM systems can quickly discard the speculative state, by extending the cache design with conditional gang-invalidation of lines whose SM bit is set.

Eager HTMs, on the other hand, must restore each cache line in the write set with the pre-transactional value that was backed up in the transaction log. The log unroll is generally done in software, by trapping to an abort handler that accesses the log base and pointer registers, and walks the log in reverse direction—those entries that were added last must be processed first. No transactional conflicts should arise during this process, as the coherence protocol ensures that the lines that belong to the write set of the aborting transaction are isolated and cannot belong to any other transaction. Because aborting is a slow process in eager HTMs, isolation over the read set is usually released as soon as the abort is triggered, as it is safe for other transactions to access it while the log is unrolled.

5 Intel TSX: TM Support in Mainstream Processors

Almost a decade after the research community regained interest in hardware implementations of TM, the world's largest semiconductor company adopted these ideas for a commercial product. The fourth generation of the Intel Core microarchitecture, commonly known by its code name *Haswell*, implements the basic mechanisms to provide programmers with a best-effort yet fast implementation of the transactional abstraction. Intel began shipping Haswell-based processors in 2013, making the Core i3/i5/i7 and Xeon v3 processor families the first chips with TM support that are available in the consumer and server markets. Given Intel's market share on mobile, desktop and servers platforms, Haswell is an important milestone towards the expansion of transactions as a synchronization primitive for multi-threaded applications.

Following the *tick* of Ivy Bridge, which shrunk the Sandy Bridge microarchitecture to the 22-nm process technology, Haswell's *tock* extends the instruction set architecture in a number of ways, from which the *Transactional Synchronization Extensions* (TSX) are certainly one of the most prominent novel features. Through the new TSX instructions, Haswell offers programmers two interfaces to exploit its ability to use optimistic concurrency in thread synchronization: *Hardware Lock*

Elision (HLE) is meant to accelerate conventional lock-based programs while maintaining legacy compatibility, while *Restricted Transactional Memory* (RTM) allows programmers to explicitly start, commit and abort transactions, thus providing a natural way of implementing transactions as a synchronization abstraction. Regardless of the TSX interface used, the same underlying hardware mechanisms are involved in the transactional execution.

5.1 Hardware Lock Elision

Hardware Lock Elision (HLE) is a legacy-compatible ISA extension aimed at extracting more thread-level parallelism from conventional lock-based programs, by using speculation to allow concurrent execution of critical sections protected by the same *mutex*. HLE comes in the form of two instruction prefixes, `XACQUIRE` and `XRELEASE`, which act as hints to delimit the boundaries of a critical section. If the processor supports TSX, each of these prefixes modifies the behaviour of the instructions that are typically used, respectively, to acquire and release a lock variable; otherwise, the prefixes are ignored and the processor executes the code without entering transactional execution, making HLE-ready binaries backwards compatible.

When the `XACQUIRE` hint is used in conjunction with the atomic instruction that attempts to acquire a free lock (e.g. `cmpxchg`), it alters its usual behaviour and prevents (*elides*) the associated write of the "busy" value. Instead, the processor enters transactional execution, adds the address of the lock to its read set and proceeds to execute the critical section speculatively. Because the globally visible value of the lock remains unchanged (i.e. "free"), other threads can read it without causing a data conflict and also enter the critical section protected by the lock. While in transactional execution, each processor leverages coherence traffic to monitor memory accesses, detecting data conflicts and rolling back as necessary.

Similarly, the `XRELEASE` prefix is paired with the store instruction that releases the lock, so that the associated write of the "free value" is again avoided. Instead, the processor attempts to commit the transactional execution. In this way, as long as threads do not perform any conflicting operations on each other's data, they can concurrently execute the critical section without unnecessary serialization due to a coarse grain lock.

If speculation fails, the processor will rollback and re-execute the critical section without using lock elision. Mutual exclusion in the re-execution of the critical section is automatically ensured, because the address of an elided lock is always added to the read set of the transaction, and thus non-transactional writes associated to lock acquisition will always cause data conflicts with all other threads that may be eliding the same lock at that time, which will be forced to rollback and also retry without elision. Therefore, code that makes use of HLE maintains the same forward progress guarantees as the underlying lock-based execution.

5.2 Restricted Transactional Memory

Unlike HLE, Restricted Transactional Memory (RTM) gives up backwards compatibility to provide programmers with a more flexible interface for transactional execution. It introduces new instructions to define transaction boundaries, `XBEGIN` and `XEND`, as well as to explicitly abort a transaction from software, `XABORT`. Transactional nesting is supported in TSX by means of flattening: the nesting level is incremented by `XBEGIN` and decremented by `XEND`, and commit is only attempted when the nesting level goes to zero.

Programmers must provide an alternative code path to the `XBEGIN` instruction, where control is transferred to in case the transaction aborts, after the processor has discarded all speculative updates, restored architectural state to appear as if the speculation never occurred, and resume execution non-transactionally. After an abort, the `EAX` register is used to communicate its cause (explicit, data conflict, internal buffer overflow, faults, etc.) to the fallback routine, as well as the 8-bit immediate taken as argument by the `XABORT` instruction. In this way, programmers may freely use the fallback path in different ways to decide the most profitable course of action, manage contention, etc. It is important to remark that according to the TSX specification, the HTM implementation is best-effort, as there are no guarantees as to whether an RTM transaction will ever successfully commit. Thus, the fallback code is entirely responsible for guaranteeing forward progress.

Figure 1 shows a simple implementation of the fallback path, which attempts to retry a transaction a number of times before acquiring a global lock to execute the transaction in serial irrevocability. This implementation shares similarities with that found in GCC's *libitm* library, since version 4.7.0. As we can see, `serial_lock` is read after the transaction has successfully started so that, when a thread enters serial irrevocable mode by acquiring the lock, it automatically causes the abort of all other running transactions due to a conflict on the lock variable, achieving a similar behaviour to what the HLE interface provides.

6 Analysing Intel TSX Performance on Haswell

In this section, we present a brief performance analysis of the Intel TSX extensions, with the purpose of shedding light into the benefits of hardware support for transactions. For this evaluation, we use a benchmark from the STAMP suite (*Stanford Transactional Applications for Multi-Processing* [13]). STAMP benchmarks are extensively used in the TM research literature. Unlike other benchmarks (e.g. SPLASH-2 [88]), the STAMP applications have been developed from scratch using coarse grain transactions, in an attempt to capture the features of future transactional workloads.

For the sake of simplicity, we focus exclusively on the application *intruder*, whose use of transactions we can see in the code snippet shown in Fig. 2. This benchmark emulates a signature-based network intrusion detection system in which packets

```
[...]
#include <immintrin.h>
#include <xmmintrin.h>
#include "spinlock.h"


void beginTransaction() {
 int nretries = 0;
 while(1) {
    /* Wait for serial irrevocable transaction to complete */
    while (!arch_read_can_lock(&serial_lock)) _mm_pause();
   ++nretries;
   unsigned status = _xbegin();
   if(status == _XBEGIN_STARTED) {
     if (!arch_read_can_lock(&serial_lock)){
     /* Started transaction but someone is executing the transaction section non-speculatively
       * (acquired fall-back lock) -> abort */
         _xabort(0xff);
     } else {
         return; /* Successfully started transaction */
     }
   }
   /* This is the beginning of the fallback path (abort handler) */
   if((status & _XABORT_EXPLICIT) &&
        _XABORT_CODE(status)==0xff &&
        !(status & _XABORT_NESTED))    {
     while (!arch_read_can_lock(&serial_lock)) _mm_pause();  /* Wait until the lock is free */
   }
   if(nretries >= MAX_RETRIES) break;     /* Too many retries, take the fallback lock */
 }
   arch_write_lock(&serial_lock);    /* Execute atomic region in serial irrevocable mode */
}

void endTransaction() {
   if (!arch_read_can_lock(&serial_lock)){      /* We were in serial irrevocable mode */
       arch_write_unlock(&serial_lock);
   } else  { /* Regular transaction, commit. */
       _xend();
   }
}
```

Fig. 1 A possible implementation of the fallback for Intel RTM

are processed in parallel and go through three phases: capture, reassembly, and detection. Transactions are used to synchronize access to the shared data structures used in the capture and reassembly phases, respectively, a simple FIFO queue and a self-balancing tree. We can see how the resulting code is simple as that of coarse-grain locks, effectively easing the task of the programmer, as opposed to the use of fine-grain locking on the queue and tree data structures.

We pick *intruder* because it exhibits several interesting characteristics. First, it comprises several transactions that access different data structures. On the one hand, its first and third transactions are used to extract an element at the head of a queue, and thus have small read and write set sizes, since it basically consist of a read-modify-write operation of the head pointer. On the other hand, its main transaction has medium-sized transactional sets—in the order of a few tens of cache lines—since it carries out most of the processing of the packet (reassembly) by traversing the tree structure. Despite its coarse granularity, its main transaction can be accommodated by the Haswell hardware without constantly causing capacity aborts. Last but not

```
void
processPackets (void* argPtr)
{
 [...]
   while (1) {
        TM_BEGIN();
        bytes = TMSTREAM_GETPACKET(streamPtr);
        TM_END();
        if (!bytes) {
            break;   /* No more packets to process */
        }
        [...]
        TM_BEGIN();
        error = TMDECODER_PROCESS(decoderPtr, bytes, (PACKET_HEADER_LENGTH + packetPtr->length));
        TM_END();
        [...]
        TM_BEGIN();
        data = TMDECODER_GETCOMPLETE(decoderPtr, &decodedFlowId);
        TM_END();
        if (data) {
            error_t error = PDETECTOR_PROCESS(detectorPtr, data);
            [...]
        }
    }
}
```

Fig. 2 Example of coarse grain transactions in *intruder*

least, intruder exhibits high levels of contention that are desirable to evaluate TM performance in less favourable conditions.

Our experiments with TSX are performed on a 3.4 GHz quad-core Intel Core i7-4770 processor with 16 GB of main memory, running Linux kernel 3.11. Each core has support for two SMT threads, but we choose to disable hyperthreading from the BIOS, in order to dedicate all available resources for speculative buffering (e.g. L1 data cache) to a single thread per core. Each core has an eight-way, 32 KB L1 data cache. Given the four hardware contexts available, we run the program with one, two and four threads. We pin one thread to each core using *pthread affinity*. The benchmark is compiled with GCC v.4.8.1, using the O3 optimization level. Since version 4.8, GCC supports the Intel RTM intrinsics, built-in functions and code generation by including the `<immintrin.h>` header and enabling the `-mrtm` flag. The `begin_transaction` and `end_transaction` functions shown in Fig. 1 are used to implement the fallback-path. The read-write spinlock implementation from `linux-3.11/arch/x86/include/asm/spinlock.h` is used to enforce serialization. Transactions are allowed to retry up to eight times before resorting to serialization via the fallback lock. For each configuration, a minimum of 20 executions are averaged to derive statistically meaningful results. Our experiments use the large input size recommended for non-simulator runs [13]: 256K traffic flows are analyzed, 10 % of which have attacks injected, where each flow has a maximum of 128 packets.

To observe the relative performance gain achieved by TSX, we consider in this experiment other two synchronization schemes that do not make use of the hardware support for transactions. We run a lock-based version of the benchmark in which transactions are implemented as critical sections protected through a *single global lock* (SGL). Additionally, we compare TSX performance against a software TM

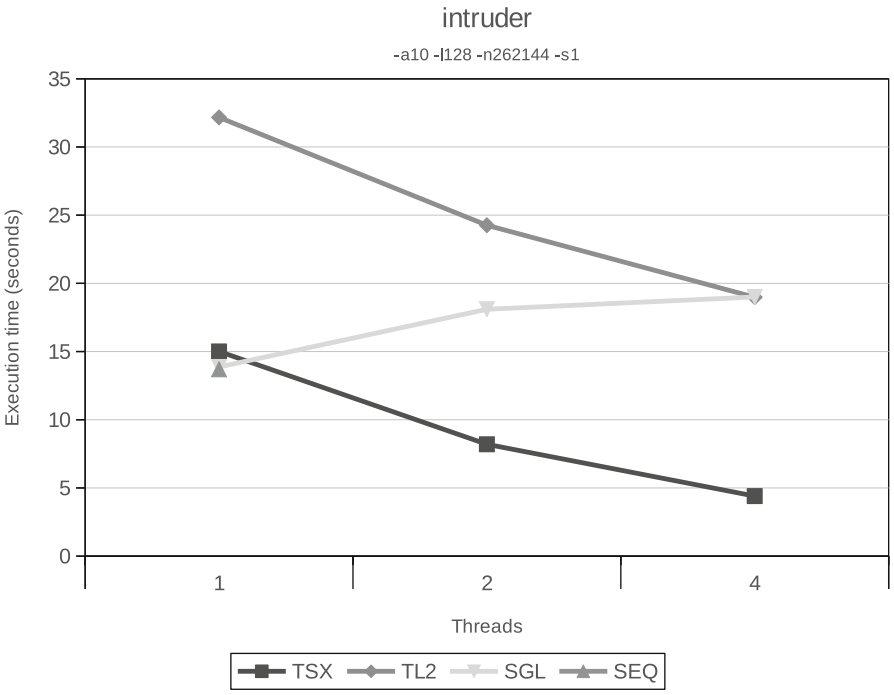

Fig. 3 Performance comparison of TSX versus TL2 and single global lock

system called *Transactional Locking II* (TL2) [26], which is distributed with STAMP. We also include a sequential flavour of the benchmark (SEQ), which is stripped of all synchronization.

Figure 3 shows the execution time of the three synchronization schemes implemented by the underlying library. The plot shows execution time (in seconds) for each of the synchronization flavours considered, and runs with one to four threads. As we can see in Fig. 3, *intruder* achieves good scalability through optimistic concurrency in spite of the coarse grain transactions used. Both hardware (TSX) and software (TL2) implementations of TM scale significantly better than a single global lock. In particular, we see how TSX is able to bring the execution time from 15 s with a single thread, down to around 4.5 when running 4 threads. As opposed to HTM and STM solutions, adding more threads in the SGL scheme does not speedup execution but rather causes a slight performance degradation: The global lock precludes all concurrency in the application, and adding more threads only makes things worse by increasing the contention on the lock variable. Given that the use of coarse grain transactions entails a similar complexity to that of single global lock, this performance comparison between SGL and two TM implementations confirms that, from the point of view of the programmer, transactions are indeed able to keep up its promise of achieving better scalability than coarse grain locks, with the same programming effort.

```
 Intel(r) Performance Counter Monitor: Intel(r) Transactional Synchronization Extensions Monitoring Utility
 Copyright (c) 2013 Intel Corporation
Num logical cores: 4
Num sockets: 1
Threads per core: 1
[...]
Detected Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz "Intel(r) microarchitecture codename Haswell"
Update every 0 seconds
 Executing "./intruder -a10 -l128 -n262144 -s1 -t4" command:
Percent attack  = 10
Max data length = 128
Num flow        = 262144
Random seed     = 1
Num attack      = 29395
Elapsed time    = 4.451121 seconds
Num found       = 29395
Core | IPC  | Instructions | Cycles  | Transactional Cycles | Aborted Cycles  | #RTM  |  Cycles/Transaction
   0   0.52         11 G       22 G        13 G (61.25%)       5569 M (24.84%) 7779 K    1765
   1   0.21       3236 M       15 G        13 G (88.90%)       3825 M (24.56%) 7177 K    1929
   2   0.21       3203 M       15 G        13 G (89.08%)       3763 M (24.17%) 7103 K    1952
   3   0.22       3348 M       15 G        13 G (88.72%)       4457 M (28.63%) 7336 K    1882
--------------------------------------------------------------------------------------------
   *   0.31         21 G       69 G        55 G (79.93%)         17 G (25.48%)   29 M    1879
Event0: RTM_RETIRED.START Number of times an RTM execution started.
Event1: RTM_RETIRED.ABORTED_MISC1 Number of times an RTM execution aborted due to various- memory events
Event2: TX_MEM.ABORT_CONFLICT Number of times a transactional abort was signaled due to a data conflict.
Event3: RTM_RETIRED.COMMIT Number of times an RTM execution successfully committed
Core | Event0  | Event1  | Event2  | Event3
   0   7087 K     619 K     470 K    5869 K
   1   7093 K     666 K     510 K    5842 K
   2   7039 K     618 K     463 K    5835 K
   3   7023 K     624 K     472 K    5800 K
-----------------------------------------------
   *     28 M    2528 K    1916 K      23 M
```

Fig. 4 PCM output for *intruder* (four threads), showing TSX performance event counts

Furthermore, the numbers obtained by TSX demonstrate that hardware implementations of transactional semantics are necessary to dramatically reduce the substantial performance overheads seen in software-only solutions, as a result of the instrumentation on every memory access within a transaction which is required to track transactional reads and writes and be able to detect conflicts. Hardware TM implementations can exhibit their full potential in those cases where most transactions are appropriately sized to avoid overflowing the hardware buffering capacity, as it is the case of *intruder* in our Haswell-based setup. Other STAMP benchmarks with larger transaction footprints may not be as well suited and exhibit substantially higher capacity-induced aborts [91].

In spite of the good speedup achieved by TSX with four threads (3.3 times faster than sequential), the scalability of *intruder* starts to deviate more and more from the ideal. This is a direct result of the increasing level of contention seen in the application: several threads attempting to capture the same packet from the FIFO queue, concurrent accesses to the dictionary while the tree is rebalanced, etc. Using the Intel Performance Counter Monitor (PCM), an open source tool, we can monitor TSX performance events in order to obtain relevant information about the execution of the program, such as the amount of contention (e.g. number of aborts due to data conflicts, `TX_MEM.ABORT_CONFLICT`). Figure 4 shows the output of PCM when running the benchmark *intruder* with four threads using TSX. The number of conflict-induced aborts may increase quickly with contention, particularly in HTM implementations

that resolve conflicts using a *requester wins* policy. Though details about its implementation have not been disclosed at the time of this writing, it is likely that Intel has adopted such simple yet livelock-prone conflict resolution strategy in Haswell, with the intent of keeping the changes in its coherence protocol to a minimum. In any case, the TSX specification clearly places on the fallback path the responsibility of providing forward progress when it detects that a transaction has failed too many times. As commented earlier, in our implementation of the abort handler all running transactions are automatically *killed* when the fallback lock is acquired due to a data conflict, thus adding to the number of contention-induced aborts.

7 An Overview of Hardware TM Research

Research in HTM design has been very active since the introduction of multicores in mainstream computing. In the early 1990s, Herlihy and Moss introduced Transactional Memory [36] as a hardware alternative to lock-based synchronization. Their main idea was to generalize the LL/SC primitives in order to perform atomic accesses not to one but to several independent memory locations, thus eliminating the need for protecting critical sections with lock variables. Almost a decade later, architects began to recover their interest in transactions at a hardware level. Rajwar and Goodman's Transactional Lock Removal (TLR) [63] was the first to apply the concept of transaction to the execution of lock-protected critical sections, merging the idea of Speculative Lock Elision (SLE) [62] with a timestamp-based conflict resolution scheme.

The early proposal by Herlihy and Moss was revived ten years later by Hammond et al., who present Transactional Coherence and Consistency (TCC) [32] as a novel coherence and consistency model that uses continuous transactional execution. The novelty of TCC stems from its "all transactions, all the time" philosophy, where transactions are the basic unit of parallel work, synchronization, memory coherence and consistency. TCC's lazy approach contains speculative updates within private caches and lazily resolves races when a committing transaction broadcasts its write-set, employing a bus to serialize transaction commits.

In contrast to Stanford's TCC, Wisconsin's LogTM [51] explores the opposite corner of the HTM design space. Moore et al. take a more evolutionary approach to transactional memory in LogTM, combining transactional support with a conventional shared memory model that enables a more gradual change towards transactional systems. LogTM is a purely eager HTM system that leverages a standard coherence protocol to perform conflict detection on individual memory requests, and makes commits fast by storing old values to a per-thread log in cacheable virtual memory, which is unrolled by a software handler in case of abort. Unlike TCC, LogTM can tolerate evictions of transactional data from caches thanks to the log, and enables conflict detection on evicted blocks through an elegant extension to the coherence protocol.

LogTM has been subsequently refined. Moravan et al. [52] introduce support for nested transactions, enabling both closed nesting with partial aborts and open nesting [55]. Open nesting is a programming language construct motivated by performance, which can improve concurrency by relaxing the atomicity guarantee. When an open nested transaction commits, the TM system releases its read and written data so that other transactions can access them without generating conflicts. Thanks to open nesting, otherwise-offending transactions can access the exposed data after the nested transaction commits, while the outer transaction still runs. This can enhance the degree of concurrency achieved by the flattening scheme found in LogTM, which enforces isolation until the outermost transaction commits. In [3], Baek et al. propose FanTM, a design that uses address signatures in hardware [14] to efficiently support transaction nesting.

Later on, Yen et al. [89] decouple transactional support from caches, removing read and write bits used for transactional book-keeping, and replacing them with *hash signatures*. This latest improvement, called LogTM-SE (*Signature Edition*), borrows the concept of Bloom filters [5] to conservatively encode a transaction's read and write set metadata. The idea of applying hash encoding towards conflict detection/thread disambiguation was first introduced into the realm of TM by Ceze et al. in [14] and [15]. The use of hash signatures for transactional book-keeping has been further explored by several authors. In [69], Sanchez et al. examine different signature organizations and hashing schemes to achieve hardware-efficient and accurate TM signatures. Quislant et al. have also studied signature organizations, basing their works in LogTM-SE. In [59], they show that locality can be exploited in order to reduce the number of bits inserted in the filter for those addresses nearby located, and reducing the number of false conflicts. More recently, the authors have studied multiset signature designs [60] which record both the read and write sets in the same Bloom filter. Yen et al. developed Notary [90], which introduces a privatization interface that allows the programmer to explicitly declare shared and private heap memory allocation, which can be used to reduce the signature size as well as the number of false conflicts arising from private memory accesses. Sanyal et al. exploit the same concept in [70], proposing a scheme that dynamically identifies thread-local variables and excludes them from the commit set, both reducing the pressure on the versioning mechanisms and improving the scalability of such phase in lazy HTMs.

In the context of signature-based eager HTM systems, Titos-Gil et al. have proposed a scheme of conflict detection at the directory level [29] that is not only capable of dealing with contention more efficiently than LogTM-SE, but can also minimize the performance degradation caused by false positives. Their solution moves transactional bookkeeping from caches to the directory, introducing separate hardware module that acts as conflict controller and works independently of the coherence controller, leaving the protocol largely unmodified.

In FASTM [46], Lupon et al. extend LogTM with a coherence protocol that enables fast abort recovery in an otherwise eager HTM, by leveraging the private cache to buffer speculative state, effectively avoiding traps to software handlers that perform log unroll as long as speculatively modified data does not overflow the private cache level. LogTM's approximation of making commits fast has also inspired OneTM [8]

[7], which uses a cache to reduce the frequency with which transactions overflow on chip resources, and proposes a simple irrevocable execution switch to handle such overflows as well as context switches, I/O or system calls inside transactions, at the cost of limited concurrency.

Bobba et al. propose TokenTM [11], another unbounded HTM design that uses the abstraction of tokens [49] to precisely track conflicts on an unbounded number of memory blocks and it handles both paging, thread migration and context switching, but incurs high state overhead. In [41], Jafri et al. improve on TokenTM and propose LiteTM, a design that maintains the same virtualization properties of TokenTM while greatly reducing the state overhead, and without sacrifying much performance. Support for transactions of unlimited duration, size and nesting depth has also been considered by proposals such as UTM [1, 44] or VTM [64], which focus on hardware schemes that provide virtualization of transactions. However, both achieve this goal by introducing large amounts of complexity in the processor and the memory subsystem. On its part, XTM [20] implements transaction virtualization support in software, using virtual memory and operating at page granularity. A similar approach is taken by Chuang et al. [19] in PTM, a page-based, hardware-supported TM design that combines transaction bookkeeping with the virtual memory system to support transactions of unbounded size, as well as to handle context switches and exceptions.

While it is not an issue for eager systems like LogTM, parallelism at commit is important for lazy systems when running applications with low contention but a large number of transactions. Transactions that do not conflict should ideally be able to commit simultaneously. The very nature of lazy conflict resolution protocols makes it difficult since only actions taken at commit time permit discovery of data races among transactions. Simple lazy schemes like the ones employing a global commit token [10] or a bus [31] do not permit such parallelism. The reason for limited parallelism at commit time is that the committing transaction has no knowledge of which other concurrently running transactions must abort to preserve atomicity. The TCC design [31] was later extended to scalable DSM architectures using directory based coherence. This proposal is called Scalable TCC (STCC) [16], and it employs selective locking of directory banks to avoid arbitration delays and thereby improve commit throughput. Pugsley et al. [58] improve over STCC by proposing even more scalable commit algorithms that reduce the number of network messages, remove the need for a centralized agent, and tackle deadlocks, livelocks and starvation scenarios.

Another approach to improve the scalability of the commit process in lazy systems has been explored by EazyHTM [82]. Tomic et al. record the information pertaining to potential conflicts, which is readily available from coherence messages during the lifetime of any transaction, and use this information at commit time to allow true commit parallelism. All potentially conflicting transactions that must be aborted would be known, and committers that have not seen races can commit in a truly parallel fashion.

Pi-TM [53] builds upon the ideas explored by EazyHTM, and leverages the concept of pessimistic self-invalidation to enable parallel lazy commits without affecting the execution in the common case. Negi et al. develop an early conflict detection—lazy conflict resolution HTM design with modest extensions to existing protocols,

which uses information regarding conflicts and performs pessimistic invalidation of potentially conflicting lines on commit and abort, enabling fast common-case execution.

FlexTM [75] also provides lazy conflict resolution by recording conflicts as they happen, using this information to enable distributed commits. Unlike EazyHTM, Shriraman et al. choose to do so in software, sacrifying progress guarantees to gain greater parallelism. Performance costs associated with software intervention and software verification challenges without watertight forward progress guarantees could limit the value of this approach. EazyHTM, on the other hand, provides parallel lazy commits in hardware and ensures forward progress, but trades off common-case performance to achieve it. FlexTM allows flexibility in policy but it does so by implementing critical policy managers in software. It provides a significant improvement in speed over software TM implementations by proposing the use of alert-on-update hardware, but the considerable cost of software intervention renders a comparison with pure HTMs moot. In the context of HTM, Shriraman and Dwarkadas [73] have also analyzed the interplay between conflict resolution time and contention management policy. They show that both policy decisions have a considerable impact on the ability to exploit available parallelism and demonstrate that conflict resolution time has the dominant effect on performance, corroborating that lazy HTMs are able to uncover more parallelism than eager approaches.

With DynTM [47], Lupon et al. introduce a cache coherence protocol that allows transactions in a multi-threaded application run either eagerly or lazily based on some heuristics like prior behavior of transactions, at the cost of adding extra complexity at level of the coherence controller. Recognizing the fact that contention is more a property of data rather than that of an atomic code block, ZEBRA [79] chooses a different dimension when combining eager and lazy policies into a HTM design, allowing per-cache-line selection of versioning and conflict resolution policies. While DynTM selects policies at the level of transactions, ZEBRA is a data-centric design which works at the same granularity of the underlying coherence infrastructure—i.e. cache lines—and therefore introduces less complexity into existing protocols. This hybrid design is able to track closely or exceed the performance of the best performing policy for a given workload, bringing together the benefits of parallel commits (inherent in traditional eager HTMs) and good optimistic concurrency without deadlock avoidance mechanisms (inherent in lazy HTMs), with little increase in complexity.

LV* [54], a proposal that utilizes snoopy coherence, allows programmer control over policy in hardware but with the constraint that all transactions in an application must use the same policy at any given time. The requirement of programmer-assisted policy change is a drawback too since the same phase of an application can exhibit different behavior with varying datasets.

The mitigation of the performance penalty associated with transaction aborts has been of interest to the HTM community. Waliullah and Stenstrom study the utility of intermediate checkpoints in lazy HTM systems [85, 86], as a means to reduce the amount of work that is discarded on abort. In their scheme transactions record conflicting addresses upon abort, and use this historical information to insert a checkpoint

before a memory reference predicted as conflicting is executed. If the transaction is squashed, it is rolled back to the checkpoint associated with the first conflicting access, rather than all the way back to the beginning. Reducing the penalty of abort was also considered by Armejach et al. [2], who propose a reconfigurable private level data cache to improve the efficiency of the version management mechanism in both eager and lazy HTMs.

Titos et al. have also analysed how the lack of effective techniques for store management results in a quick degradation in the performance of eager HTM systems with increasing contention and, thus, lends credence to the belief that eager designs do not perform as well as their lazy counterparts when conflicts abound [80]. The authors present two simple ways to improve handling of speculative stores which yield substantial improvements in execution time when running applications with high contention, allowing eager designs to exceed the performance of lazy ones.

The applications of data forwarding and value prediction for conflict resolution have also been explored in the context of eager HTM systems. Pant et al. [56, 57] observe that shared-conflicting data is often updated in a predictable manner by different transactions, and propose the use of value prediction in order to capture this predictability and increase overall concurrency by satisfying loads from conflicting transactions with predicted values, instead of stalling. In DATM [66], Ramadan et al. investigate the advantages of value forwarding for speculative resolution of true data conflicts amongst concurrent transactions. DATM is an eager system that discovers and tracks the data dependencies amongst concurrent transactions, allowing writer transactions to proceed in the presence of other conflicting transactional accessors, and reader transactions to obtain uncommitted data produced by a concurrent transaction, while still enforcing a legal serialized order that preserves consistency.

Hardware TM systems can suffer a series of pathological behaviours that negatively affect performance. Bobba et al. explore HTM design space, identifying how some of these undesirable scenarios [10] affect each kind of system depending on the choice of policies for version and conflict management. Some pathologies such as starvation have been further analysed and resolved in other subsequent works [87]. Other pathologies that affect HTM performance have been the topic of several studies. Volos et al. [84] investigate the interaction of transactional memory implementations and lock-based code, and discover other problematic scenarios that may arise in these circumstances. False sharing, another undesired situation that may arise in multi-threaded codes, becomes even a bigger problem when it occurs in conjunction with hardware transactional memory [51] due to the detection of conflicts at a cache line granularity. Tabba et al. [78] propose a mechanism that takes the concepts of coherence decoupling [39] and value prediction, and combines them to mitigate the effects of coherence conflicts in transactions. The granularity of conflict detection in HTM has also been the subject of the works by Khan et al. [42], whose HTM proposal is able to detect conflicts at the level of objects—instead of cache lines—which leads to a novel commit scheme as well as an elegant solution to the problem of version management virtualization.

Another kind of pathological behaviour affecting HTM performance happens when concurrent operations on data structures that are not semantically conflicting—such as two insertions in two different buckets of a hash table—result in conflicting transactions because of updates on auxiliary program data—e.g. the *size* field. Inspired by instruction replay-based mechanisms [25], Blundell at al. propose RetCon [9], a hardware mechanism that eliminates the performance impact of such spureous transactional conflicts. RetCon tracks the relationship between input and output values symbolically and uses this information to transparently repair the output state of a transaction at commit.

Ramadan et al. have examined the architectural features necessary to support HTM in the Linux kernel for the x86 architecture [65, 68]. They propose MetaTM, an HTM model that contains features that enable efficient and correct interrupt handling for an x86-like architecture. Using TxLinux—a Linux kernel modified to use transactions in place of locking primitives in several key subsystems—they quantify the effect of architectural design decisions on the performance of such a large transactional workload. TxLinux, based on the Linux 2.4 kernel and thus characterized by its simple, coarse-grained synchronization structure, is used by Hoffman et al. in [38] to show that a minimal subset of TM features supported in hardware can simplify synchronization, provide comparable performance to fine-grained locking and handle overflows. The challenge of operating system (OS) support in HTM is also addressed Wang et al. [71] and Tomic et al. [81]. DTM [71] proposes a hardware-based solution that fully decouples transaction processing from caches, while HTM-OS [81] leverages the existing OS virtual memory mechanisms to support unbounded transaction sizes and provide transaction execution speed that does not decrease when transaction grows. A related challenge that has been addressed in the HTM literature is the support of input/output operations within transactions: Lui et al. [45] analyse this problem and propose an HTM system that supports I/O within transactions by means of partial commits, using commit-locks and blocking/waking-up of transactional threads.

The applicability of hardware transactional memory (HTM) has also been considered in the context of dynamic memory management. Dragojevic et al. [28] demonstrate that HTM can be used to simplify and streamline memory reclamation for practical concurrent data structures. The use of HTM to aid lightweight dynamic language runtimes in evolving more capable and robust execution models while maintaining native code compatibility has been studied too. Using a modified Linux kernel and a Python interpreter, Riley at al. [67] explore the lack of thread safety in native extension modules and use features found in an HTM implementation to address several issues that impede to the effective deployment of dynamic languages on current and future multicore and multiprocessor system.

8 Conclusions

Following the recent inclusion of hardware support for Transactional Memory in commodity multicore processors [91] and commercial mainframes [40], the time has come for architects and programmers of datacenters to ponder the new opportunities that may unfold in the coming years. This chapter examines the state-of-the-art of Transactional Memory, paying special attention to its hardware implementations (*Hardware Transactional Memory* or HTM). Transactions not only address one the key challenges of the multicore era, i.e. the trade-off between programming ease and performance, but also bring about other important benefits such as better code composability and fault tolerance. For these reasons, parallel software developed for large-scale clusters may also find in Transactional Memory an attractive programming model to unlock the full potential of the multicore processors that power a datacenter, while improving aspects that impact the total cost of ownership such as server utilization or code maintainability.

Acknowledgements This work was supported by the Spanish MINECO, as well as European Commission FEDER funds, under grant TIN2012-38341-C04-03.

References

1. C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th Symposium on High-Performance Computer Architecture*, pages 316–327, 2005.
2. Adria Armejach, Azam Seydi, Rubén Titos-Gil, Ibrahim Hur, Adrián Cristal, Osman Unsal, and Mateo Valero. Using a reconfigurable l1 data cache for efficient version management in hardware transactional memory. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, 2011.
3. Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Making nested parallel transactions practical using lightweight hardware support. In *Proceedings of the 24th International Conference of Supercomputing*, pages 61–71, 2010.
4. Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 115–126. 2008.
5. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.
6. Colin Blundell, E Christopher Lewis, and Milo Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), 2006.
7. Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Unrestricted transactional memory: Supporting i/o and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, 2006.
8. Colin Blundell, Joe Devietti, E Christopher Lewis, and Milo Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 24–34, 2007.
9. Colin Blundell, Arun Raghavan, and Milo M.K. Martin. RETCON: transactional repair without replay. In *Proceedings of the 37th International Symposium on Computer Architecture*, pages 258–269, 2010.

10. Jayaram Bobba, Kevin E. Moore, Luke Yen, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 81–91, 2007.
11. Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 81–91, 2008.
12. Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 69–80, 2007.
13. Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE Intl. Symposium on Workload Characterization*, pages 35–46, 2008.
14. Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 227–238, 2006.
15. Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: bulk enforcement of sequential consistency. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 278–289, 2007.
16. Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A scalable, non-blocking approach to transactional memory. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, pages 97–108, 2007.
17. Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A high-performance Sparc CMT processor. *IEEE Micro*, 29(2):6–16, 2009.
18. Ben Chelf. Ensuring code quality in multi-threaded applications. http://www.coverity.com/library/pdf/coverity_multi-threaded_whitepaper.pdf.
19. Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *Proceedings of the 12th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 347–358, 2006.
20. JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. Tradeoffs in transactional memory virtualization. In *Proceedings of the 12th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 371–381, 2006.
21. Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, and Dan Grossman. Asf: Amd64 extension for lock-free data structures and transactional memory. In *Proceedings of the 43rd International Symposium on Microarchitecture*, pages 39–50, 2010.
22. Cliff Click. Azul's experiences with hardware transactional memory, 2009. http://sss.cs.purdue.edu/projects/tm/tmw2010/talks/Click-2010_TMW.pdf.
23. Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 336–346, 2006.
24. James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the 1stInternational Symposium on Code Generation and Optimization (CGO)*.
25. Rajagopalan Desikan, Simha Sethumadhavan, Doug Burger, and Stephen W. Keckler. Scalable selective re-execution for edge architectures. In *Proceedings of the 11th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 120–132, 2004.

26. David Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 19th Intl. Symposium on Distributed Computing*, 2006.
27. Ivan Dobos *et al.* IBM zEnterprise EC12 Technical Guide, February 2013. http://www.redbooks.ibm.com/redbooks/pdfs/sg248049.pdf.
28. Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. On the power of hardware transactional memory to simplify memory management. In *Proceedings of the 30th International Symposium on Principles of Distributed Computing*, pages 99–108, 2011.
29. J. Rubén Titos Gil, Manuel E. Acacio, and José M. García. Efficient eager management of conflicts for scalable hardware transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 24(1):59–71, 2013.
30. Tom Groenfeldt. Software programmers lag behind hardware developments, 2011. http://blogs.forbes.com/tomgroenfeldt/2011/04/21/software-programmers-lag-be hind-hardware-developments/.
31. Lance Hammond, Brian D. Carlstrom, Vicky Wong, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6), 2004.
32. Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 102–113, 2004.
33. Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, alan gara, George Chiu, Peter Boyle, Norman Chist, and Changhoan Kim. The ibm blue gene/q compute chip. *IEEE Micro*, 32(2):48–60, March 2012.
34. Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan & Claypool, 2010.
35. Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Symposium on Principles of Distributed Computing*, pages 92–101, 2003.
36. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, 1993.
37. Owen S. Hofmann, Donald E. Porter, Christopher J. Rossbach, Hany E. Ramadan, and Emmett Witchel. Solving difficult HTM problems without difficult hardware. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, 2007.
38. Owen S. Hofmann, Christopher J. Rossbach, and Emmett Witchel. Maximum benefit from a minimal HTM. In *Proceedings of the 14th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 145–156, 2009.
39. Jaehyuk Huh, Jichuan Chang, Doug Burger, and Gurindar S. Sohi. Coherence decoupling: making use of incoherence. In *Proceedings of the 11th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 97–106, 2004.
40. Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for IBM System z. In *Proceedings of the 45th International Symposium on Microarchitecture*, pages 25–36, 2012.
41. Syed Ali Raza Jafri, Mithuna Thottethodi, and T. N. Vijaykumar. LiteTM: Reducing transactional state overhead. In *Proceedings of the 16th Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.
42. Behram Khan, Matthew Horsnell, Ian Rogers, Mikel Luján, Andrew Dinn, and Ian Watson. An object-aware hardware transactional memory. In *Proceedings of the 10th International Conference on High Performance Computing and Communications*, pages 93–102, 2008.
43. Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 209–220, 2006.

44. Sean Lie. Hardware support for unbounded transactional memory. Master's thesis, 2004. Massachusetts Institute of Technology.
45. Yi Liu, Xin Zhang, He Li, Mingxiu Li, and Depei Qian. Hardware transactional memory supporting I/O operations within transactions. In *Proceedings of the 10th International Conference on High Performance Computing and Communications*, pages 85–92, 2008.
46. Marc Lupon, Grigorios Magklis, and Antonio González. FASTM: A log-based hardware transactional memory with fast abort recovery. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 293–302, 2009.
47. Marc Lupon, Grigorios Magklis, and Antonio González. A dynamically adaptable hardware transactional memory. In *Proceedings of the 43rd International Symposium on Microarchitecture*, pages 27–38, 2010.
48. Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th Intl. Symposium on Distributed Computing*, 2005.
49. Milo M.K. Martin. *Token Coherence*. PhD thesis, CS Dept., Univ. of Wisconsin-Madison, 2003.
50. Austen McDonald, JaeWoong Chung, D. Carlstrom Brian, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 53–65, 2006.
51. Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th Symposium on High-Performance Computer Architecture*, pages 254–265, 2006.
52. Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in LogTM. In *Proceedings of the 12th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 359–370, 2006.
53. Anurag Negi, J. Rubén Titos Gil, Manuel E. Acacio, José M. García, and Per Stenström. Pi-tm: Pessimistic invalidation for scalable lazy hardware transactional memory. In *Proceedings of the 18th Symposium on High-Performance Computer Architecture*, pages 141–152, 2012.
54. Anurag Negi, M.M. Waliullah, and Per Stenstrom. LV*: A low complexity lazy versioning HTM infrastructure. In *Proceedings of the Intl. Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS 2010)*, pages 231–240, 2010.
55. Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Proceedings 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 68–78, 2007.
56. Salil Pant and Gregory Byrd. Extending concurrency of transactional memory programs by using value prediction. In *Proceedings of the 6th ACM conference on Computing Frontiers*, pages 11–20, 2009.
57. Salil Pant and Gregory Byrd. Limited early value communication to improve performance of transactional memory. In *Proceedings of the 23rd International Conference of Supercomputing*, pages 421–429, 2009.
58. Seth H. Pugsley, Manu Awasthi, Niti Madan, Naveen Muralimanohar, and Rajeev Balasubramonian. Scalable and reliable communication for hardware transactional memory. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 144–154, 2008.
59. Ricardo Quislant, Eladio Gutierrez, and Oscar Plata. Improving signatures by locality exploitation for transactional memory. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 303–312, 2009.
60. Ricardo Quislant, Eladio Gutierrez, and Oscar. Plata. Multiset signatures for transactional memory. In *Proceedings of the 25th International Conference of Supercomputing*, pages 43–52, 2011.

61. Ravi Rajwar and Martin Dixon. Intel transactional synchronization extensions, 2012. Intel Developer Forum (IDF2012).
62. Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 294–305, 2001.
63. Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 5–17, 2002.
64. Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 494–505, 2005.
65. Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. MetaTM/TxLinux: transactional memory for an operating system. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 92–103, 2007.
66. Hany E. Ramadan, Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Dependence-aware transactional memory. In *Proceedings of the 41st International Symposium on Microarchitecture*, pages 246–257, 2008.
67. Nicholas Riley and Craig Zilles. Hardware transactional memory support for lightweight dynamic language evolution. In *Dynamic Language Symposium*, 2006.
68. Christopher J. Rossbach, Hany E. Ramadan, Owen S. Hofmann, Donald E. Porter, Aditya Bhandari, and Emmett Witchel. TxLinux and MetaTM: transactional memory and the operating system. *Communications of the ACM*, 51:83–91, 2008.
69. Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of the 40th International Symposium on Microarchitecture*, pages 123–133, 2007.
70. Sutirtha Sanyal, Adrián Cristal, Osman S. Unsal, Mateo Valero, and Sourav Roy. Dynamically filtering thread-local variables in lazy-lazy hardware transactional memory. In *Proceedings of the 11th International Conference on High Performance Computing and Communications*, pages 171–179, 2009.
71. Wang Shaogang, Dan Wu, Zhengbin Pang, and Xiaodong Yang. DTM: Decoupled hardware transactional memory to support unbounded transaction and operating system. In *Proceedings of the 38th International Conference on Parallel Processing*, pages 228–236, 2009.
72. Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.
73. Arrvindh Shriraman and Sandhya Dwarkadas. Refereeing conflicts in hardware transactional memory. In *Proceedings of the 23rd International Conference of Supercomputing*, pages 136–146, 2009.
74. Arrvindh Shriraman, Virendra J. Marathe, Sandhya Dwarkadas, Michael L. Scott, David Eisenstat, Christopher Heriot, William N. Scherer III, and Michael F. Spear. Hardware acceleration of software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.
75. Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 139–150, 2008.
76. Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. 30(3), 2005.
77. Fuad Tabba, Cong Wang, James R. Goodman, and Mark Moir. NZTM: Nonblocking, zero-indirection transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2007.
78. Fuad Tabba, Andrew W. Hay, and James R. Goodman. Transactional conflict decoupling and value prediction. In *Proceedings of the 25th International Conference of Supercomputing*, pages 33–42, 2011.

79. Rubén Titos-Gil, Anurag Negi, Manuel E. Acacio, Jose M. Garcia, and Per Stenstrom. Zebra: A data-centric, hybrid-policy hardware transactional memory design. In *Proceedings of the 25th International Conference of Supercomputing*, pages 53–62, 2011.
80. Ruben Titos-Gil, Anurag Negi, Manuel E. Acacio, Jose M. Garcia, and Per Stenstrom. Eager beats lazy: Improving store management in eager hardware transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints):1, 2012.
81. Sasa Tomic, Adrian Cristal, Osman Unsal, and Mateo Valero. Hardware transactional memory with operating system support, HTMOS. In *Proceedings of the 13th European Conference on Parallel Processing (Euro-Par)*, pages 8–17, 2007.
82. Sasa Tomic, Cristian Perfumo, Chinmay Kulkarni, Adria Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. EazyHTM: Eager-lazy hardware transactional memory. In *Proceedings of the 42nd International Symposium on Microarchitecture*, pages 145–155, 2009.
83. Hans Vandierendonck and Tom Mens. Averting the next software crisis. *IEEE Computer*, 44:88–90, 2011.
84. Haris Volos, Neelam Goyal, and Michael M. Swift. Pathological interaction of locks with transactional memory. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, 2008.
85. M. M. Waliullah. Efficient partial roll-backing mechanism for transactional memory systems. *Transactions on high-performance embedded architectures and compilers*, 3:256–274, 2011.
86. M.M. Waliullah and Per Stenstrom. Reducing roll-back overhead in transactional memory systems by checkpointing conflicting accesses. In *Proceedings of the 22nd International Parallel and Distributed Processing Symposium*. 2008.
87. M. M. Waliullah and Per Stenstrom. Schemes for avoiding starvation in transactional memory systems. *Concurrency and Computation: Practice and Experience*, 21:859–873, 2009.
88. Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, 1995.
89. Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, pages 261–272, 2007.
90. Luke Yen, Stark C. Draper, and Mark D. Hill. Notary: Hardware techniques to enhance signatures. In *Proceedings of the 41st International Symposium on Microarchitecture*, pages 234–245, 2008.
91. Richard Yoo, Christopher Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel transactional synchronization extensions for high performance computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2013.