Transactional Memory

Håkan Grahn

School of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
Hakan.Grahn@bth.se, http://www.ipd.bth.se/~hgr/

Abstract

Current and future processor generations are based on multicore architectures where the performance increase comes from an increasing number of cores on a chip. In order to utilize the performance potential of multicore architectures the programs also need to be parallel, but writing parallel programs is a non-trivial task. Transactional memory tries to ease parallel program development by providing atomic and isolated execution of code sequences, enabling software composability and protected access to shared data. In addition, transactional memory has the ability to execute atomic code sequences in parallel as long as no data conflicts occur. Transactional memory implementation proposals exist for both hardware and software, as well as hybrid solutions. This special issue on transactional memory introduces transactional memory as a concept, presents an overview of some of the most important approaches so far, and finally, includes five articles that advances the state-of-the-art in transactional memory research.

Key words: Multiprocessors, Parallel Programming, Concurrency, Transactions, Synchronization

1. Introduction

Today we are in the middle of a paradigm shift in the computer industry. Previous processor generations were based on uni-processor technology, while current and future processor generations are based on multicore architectures where the performance increase are expected to mainly come from an increasing number of cores on a chip [72]. However, the software needs to be parallel as well as scalable in order to harvest the multicore performance potential [5, 68, 97].

Parallel programming in the multicore era is a challenging task. Writing parallel applications can be cumbersome, fault-prone, and take a lot of time and effort. Partitioning the application into concurrent threads and ensuring that the application is properly synchronized introduce correctness issues, e.g., race conditions, deadlocks, and priority inversion problems. The introduction of synchronizations may also lead to unnecessary dependencies in the code that can have a severe negative performance impact. Thus, novel approaches are needed to ease the development of parallel applications.

Transactional memory [49, 90, 56, 12] has been proposed as an alternative to the traditional lock-based approach to express and manage concurrency. During the last few years we have seen an increasing interest in programming languages, run-time systems, and hardware to support transactional memory, speculative concurrency, and failure atomicity.

This special issue of the *Journal of Parallel and Distributed Computing* covers a wide range of aspects and captures the state-of-the-art of an emerging area. The introduction to the special issue provides an introduction to transactional memory as well as an overview of some of the most important transactional memory proposals up to late 2009. A thorough presentation of transactional memory proposals up to mid 2006 is found in [56]. The other papers in the special issue constitute a selection of high-quality papers advancing the state-of-the-art in transactional memory research.

Thirteen papers were submitted to the special issue. Two of those papers were judged as out of scope and the other eleven papers were sent out for review. All papers were reviewed by three, and sometimes four, reviewers. Many of the papers were reviewed by people from both industry and academia. Based on the reviewers' comments, we decided to accept five high-quality papers for inclusion in the special issue. The papers cover several different and important aspects of transactional memory.

The first paper, *"Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency"* by Usui, Behrends, Evans, and Smaragdakis, addresses how to combine different synchronization approaches. Their approach is based on adaptive techniques that dynamically determine whether a critical section is best executed using locks or transactions. They provide novel techniques for on-line cost-benefit analysis and low-overhead statistical measurements, as well as a full compiler implementation of their techniques.

The next two papers address different aspects of trans-

actional memory in embedded systems. *"Lightweight Transactional Memory Systems for NoCs Based Architectures: Design, Implementation and Comparison of Two Policies"* by Meunier and Pétrot evaluates how hardware transactional memory can be used in an embedded system with write-through caches. The paper presents a detailed hardware transactional memory design for write-through caches, as well as a comparison of both implementation cost and performance of hardware transactional memory solutions for write-through and write-back caches.

"Embedded-TM: Energy and Complexity-Effective Hardware Transactional Memory for Embedded Multicore Systems" by Ferri, Wood, Moreshet, Bahar, and Herlihy also addresses implementation of hardware transactional memory in embedded systems. They propose and evaluate three alternative hardware transactional memory implementations and three contention management schemes in terms of energy, performance, and complexity.

The fourth paper, *"Extensible Transactional Memory Testbed"* by Harmanci, Gramoli, Felber, and Fetzer, addresses how to evaluate the correctness of a transactional memory implementation. More specifically, they propose a general framework, TMUNIT, for evaluation how different transactional memory implementations fulfil certain correctness and semantical properties. TMUNIT also provides a domain specific language for writing workloads.

Finally, *"Implementation Tradeoffs in the Design of Flexible Transactional Memory Support"* by Shriraman, Dwarkadas, and Scott, presents a high-performance framework, FlexTM (FLEXible Transactional Memory), for implementing transactional memory systems. Four hardware mechanisms are proposed in order to decouple hardware-based conflict detection from software controlled conflict resolution policy. Further, different hardware-software implementation alternatives are presented and evaluated.

The rest of the paper is organized as follows. Section 2 presents an introduction to transactional memory in general and the basic concepts. Then, Section 3 presents some key concepts for basic design and implementation of hardware and software transactional memory systems. In Section 4, Section 5, and Section 6 some of the most important hardware-based, software-based, and hybrid transactional memory systems, respectively, are presented. Finally, some concluding remarks are presented in Section 7.

2. Transactional Memory Concepts and Properties

2.1. Transactional Memory

Transactional memory [49, 90, 56, 12], has attracted a substantial amount of research focus during the last decade. Transactional memory tries to ease the development of parallel programs by providing primitives to execute atomic code sequences concurrently as long as no data conflicts occur. Thus, fine-grain concurrency and data access is enabled, and transactional memory has the potential to achieve higher performance than in traditional lock-based approaches [49]. Today, many argue that the main advantage of transactional memory is to provide software composability and enable software composition, see e.g., [56].

Transactional memory can be implemented both in hardware or software, as well as in various hybrid approaches such as virtualized hardware transactional memory systems, hardware accelerated software transactional memory systems, and systems that dynamically switch between hardware and software execution modes. Common issues to address for all transactional memory proposals are how to maintain speculative data (data versioning), and how to detect and resolve conflicts during the execution of concurrent transactions.

2.2. General Transactional Memory Terminology

Transactions is a general concept and has been used a long time. A transaction generally has three phases:

1. Begin the transaction. A snap-shot of the execution state is taken, which will be needed if the transaction is aborted.
2. Execution of one or several operations/actions/tasks (the terminology varies). The effects of these operations are not visible outside the transaction during the execution of the transaction.
3. Commit or abort the transaction. In case of a commit, the result of the transaction and its associated operations/actions are made visible to the rest of the system. In case of an abort, the execution is rolled-back to the start of the transaction.

In transactional memory systems, a transaction is a finite code sequence that satisfies the *atomicity* and *isolation* properties [49, 41, 70].

- **Atomicity:** Either the whole transaction is executed (committed) or none of it is done (aborted), often referred to as the "all or nothing" property.
- **Isolation:** Individual memory updates within an ongoing transaction is not visible outside the transaction. When the transaction commits, all memory updates are made visible to the rest of the system.

The first proposal where caches are used to buffer speculative updates along with old values for atomic code sequences and then let the coherence protocol detect conflict is presented by Knight in [52], thus providing the ground for future hardware transactional memory proposals. Ideas similar to the definition of transactional memory can also be found in the Oklahoma Update system [96]. They propose to update multiple shared variables atomically, supporting the "all or nothing" property. Further, the storage system of the IBM 801 implemented support for transactions in hardware [17].

In order to continue our description of transactional memory properties we need to (informally) describe what me mean with the following concepts:

- **Read-Set:** The set of data items (memory locations) that are read by a transaction.
- **Write-Set:** The set of data items (memory locations) that are written by a transaction.
- **Commit:** When a transaction successfully completes, we say that the transaction commits. When the transaction commits, all new values for the data items in the transaction's write-set are made visible to the rest of the system.
- **Abort:** Abort means that the transaction fails, usually as a result of a conflict. When a transaction aborts it must restore its initial state, i.e., reset all data items in the transaction's write-set to the value they had when the transaction began.
- **Conflict:** Two concurrent transactions is said to conflict if one transaction's write-set overlaps with the other transaction's read or write-set. In case of a conflict, one of the transactions needs to abort.

The union of the read-set and the write-set of a transaction is sometimes called the *footprint* [100] of the transaction, i.e., the set of data items that have be accessed within the transaction.

2.3. Data Version Management

A transactional memory system requires that *data version management*, in software transactional memory systems often called *update strategy* or *update policy*, is implemented. It is necessary so that the system can maintain both old values of data items (i.e., valid when the transaction starts) that are needed if the transaction aborts, and new values of data items (uncommitted values written during the execution of a transaction) that are needed when the transaction commits. For large transactions, the state that is necessary to maintain may be large.

The two basic approaches are *lazy* and *eager* data versioning. The main principle behind *lazy data versioning*, also called *deferred update*, is that all writes performed within a transaction are buffered in a write buffer or stored in a local copy of an object until the transaction commits. When the transaction commits, the values from the write buffer or the local object copy are written to memory and the write buffer is emptied. If the transaction aborts, the old values are still in memory and we only need to flush the write buffer or discard local object copies. This approach favours fast aborts at the price of slower commits.

The main principle behind *eager data versioning*, also called *direct update*, is that all writes within a transaction are performed directly in memory, and the old values of the data items are stored in an undo log (a.k.a. transaction log). When the transaction commits, the memory already contains the new values and the undo log is flushed. If the transaction aborts, the old values of the data items must be restored, i.e., fetched from the undo log and rewritten to memory. This approach favours fast commits at the price of slower aborts.

2.4. Concurrency Control

A transactional memory system must also implement *concurrency control*, i.e., *conflict detection and resolution*, in order to detect and handle conflicts between concurrent transactions accessing (and at least one updates) the same variable(s). The conflicts can be read-write, write-read, and write-write conflicts between accesses to the same data item(s). In order to detect conflicts, each transaction needs to keep track of its read-set and write-set. Similarly to data versioning, there are two basic approaches to handle conflict detection and resolution (i.e., maintain concurrency control): *lazy* (also referred to as late, optimistic, or commit) and *eager* (also referred to as early, pessimistic, or encounter) conflict detection and resolution.

Lazy conflict detection and resolution is based on the principle that the system detects conflicts when a transaction tries to commit, i.e., the conflict itself and the detection of it occur at different points in time. Then, the write-set of the committing transaction is compared to the read- and write-sets of other transactions. In case of a conflict, the committing transaction succeeds and the other transactions abort. The advantages of lazy conflict detection are, e.g., potentially fewer conflicts and enabling bulk communication. However, conflicts are detected late, which may result in more work to undo.

Eager conflict detection and resolution is based on the principle that the system checks for conflicts during each load and store, i.e., the system detects a conflict directly when it occurs. In hardware this is done by tracking coherence lookups done by the cache coherence protocol, and in software this can be done by using locks and/or version numbers. Conflicts are detected early in eager conflict detection schemes, which usually results in less work to undo in case of a conflict. However, eager conflict detection schemes may result in more aborts in some cases and often require fine-grain communication.

The *granularity* of conflict detection is a key design decision in a transactional memory system, since the read- and write-sets are maintained for data items at the conflict detection granularity. Conflict detection is usually done at the *word granularity*, *cache line granularity*, or *object granularity*.

- **Word granularity:** Tracking the read and write-sets at the word granularity maintains the read- and write-sets exactly, i.e., no false sharing [29, 28] occurs. However, this approach introduces higher overhead in terms of time and space (state information).
- **Cache line granularity:** Tracking the read and write-sets at the cache line granularity is suitable mainly for hardware and hybrid transactional memory implementations, and can leverage on existing coherence mechanisms. However, there is a risk for false sharing, which may lead to unnecessary aborts.

- **Object granularity:** Tracking the read and write-sets at the object granularity matches the programmer's reasoning, has low overhead in terms of time and space, and is suitable for software and hybrid transactional memory implementations. However, there is a risk for false sharing on large objects, which may lead to unnecessary aborts.

2.5. Nested Transactions

A nested transaction is a transaction that have one or several other transactions inside of it. The main motivation for supporting nesting is software composability. A piece of software executing within a transaction may invoke a routine in another module, which may contain transactions as well.

Closed nested transactions extend atomicity and isolation of an inner transaction until the outermost (top-level) transaction commits. Some hardware transactional memory systems implement closed nesting by flattening nested transactions into the outermost level [41, 4, 70]. In such systems an abort of an inner transaction may cause a complete abort to the beginning of the outermost transaction, which may result in low performance. A partial abort of only the inner transaction may result in higher performance by avoiding the abortion of the, possibly longer, outermost transaction [71].

Open nested transactions, in contrast, allow a committing inner transaction to release isolation immediately. This will potentially result in higher parallelism and expressiveness [71]. However, there is also a higher cost in terms of more complex hardware and software.

2.6. Strong and Weak Atomicity

There is a question whether non-transactional code can, from outside the transaction, read non-committed updated values within an ongoing transaction. *Weak atomicity* [9, 10] (sometimes called *weak isolation*) is when non-transactional code can read non-committed updates, while *strong atomicity* [9, 10] (sometimes called *strong isolation*) is when non-committed updates cannot be read from the outside of a transaction.

Strong atomicity provides a simple and intuitive model to the programmer, but may be difficult to implement efficiently. In contrast, weak atomicity may be easier to implement efficiently but provides a less intuitive model to the programmer since shared data may be accessed from outside transactions that were supposed to be atomic. However, it is important to notice that applications that assume and execute correctly under weak atomicity do not necessarily execute correctly under strong atomicity as shown in [10].

Strong atomicity is relatively easy to implement in hardware transactional memory systems since all reads and writes to a memory block are tracked by the coherence protocol. As a result, all hardware transactional memory proposals in this survey support strong atomicity. However, strong atomicity is more difficult to implement in software transactional memory systems. Most software transactional memory proposals so far have only supported weak atomicity, but recent work, e.g., [13, 91, 6, 2], show how some of these short-comings can be addressed.

3. Transactional Memory Implementation

This section describes some generic implementation principles of transactional memory. The focus is on the general mechanisms necessary and the general way of working. Then, Section 4 and Section 5 describe specific hardware and software transactional memory proposals, respectively, that have been published.

3.1. Implementation of Hardware Transactional Memory

One of the most important argument in favour of implementing transactional memory systems in hardware is performance. They have, in general, higher performance than both traditional lock-based approaches and software-based implementations of transactional memory. A second important argument of hardware transactional memory systems are binary compatibility. Ideally, a hardware transactional memory system works with all binaries and libraries without any need for recompilation.

The most prominent drawback of hardware transactional memory systems is the limitation in hardware resources. A certain hardware implementation will have a limited amount of resources to handle transactions, e.g., a fixed maximum size of the read- and write-sets. This limitation has impact on, and may cause problems for, unbounded (long) transactions as well as nested transactions. This is the reason why most hardware transactional memory systems have some virtualization support, either in time or space, as will be described later.

A hardware transactional memory does, in general, rely on the cache hierarchy and the coherence protocol to implement version handling and conflict detection. In addition, it requires at least three hardware mechanisms:

1. **A check-pointing mechanism.** The check-pointing mechanism stores the processor state (e.g., program counter and register values) at the start of the transaction. The check-pointed state is needed if the transaction is aborted and needs to restart.
2. **Data version management.** The system needs to store both old data values (restored at aborts) and new values (needed at commits). Lazy version management schemes store the new data values in a hardware write-buffer. Eager version management schemes copy old values to an undo log in hardware and new values are written to the cache.
3. **Conflict detection and resolution.** The read- and write-sets of a transaction are tracked by associating read and write bits with either each word or each cache line in the cache. Since the coherence protocol usually works at the cache line granularity, adding read and write-bits with each cache line

is a minor extension. The read- and write-sets are updated as a result of regular cache accesses. The coherence protocol actions in combination with the read and write-bits can then detect transaction conflicts. All read and write-bits are cleared when a transaction aborts or commits.

3.2. Virtualization of Hardware Transactional Memory

Pure hardware transactional memory implementations have only a fixed and platform specific number of resources, which may cause problems when these resources are exhausted. The limitations can be in both space and time. Space related limitations to handle may be, e.g., a limited number of entries in the undo log or write buffer [49], support for only a limited number of updates [96], or no support for nested transactions [70]. Time related limitations to handle include, e.g., context switches, process migration, and other types of interrupts such as page faults. Further, in many cases the programmer must know these limitations when programming transactional memory.

One solution to the problems mentioned above is to use virtualization techniques in order to enable execution of transactions with large footprints and also make transactions survive various types of interrupts, e.g., context switches and page faults. Examples of hardware transactional memory proposals supporting virtualization are UTM [4], VTM [79], and XTM [19].

3.3. Signature-Based Transactional Memory

An alternative approach to implement unbounded transactional memory is to use *signatures* [14, 15, 103, 13, 87]. Signatures are data structures that can store the access information, i.e., addresses and read-/write-sets, for a thread in a compact form using Bloom filters [7]. Since a signature uses hashing to encode the addresses, a superset representation of the original addresses is created. Thus, in contrast to other pure hardware transactional memory proposals, a signature-based transactional memory uses an *inexact* representation of the read- and write-sets.

In a signature-based hardware transactional memory system, the read- and write-bits in the caches are replaced by hardware implemented Bloom filters [7]. Various combinations of Bloom [7] and Cockoo hashing [73, 74], called *Cuckoo-Bloom Signatures* have also been evaluated [87]. One filter encodes the read-set and one encodes the write-set. The filters are updated at each load and store access, as well as checked on coherence messages from other nodes.

Advantages of signatures include, e.g., decoupling the tracking of the read-/write-sets from the cache design, encoding of unbounded read- and write-sets into a fixed-size hardware structure, simplified nesting (easy to merge Bloom signatures), and signatures can be swapped to memory or sent to other processors easily. Disadvantages include, e.g., inexact read-/write-set information and inexact operations on them. This may lead to false conflicts (a.k.a. false positives), possibly resulting in lower performance.

3.4. Implementation of Software Transactional Memory

There are several arguments in favour of implementing transactional memory systems in software. In general, a software transactional memory implementation runs on conventional systems and do not require any changes to the hardware. At the same time, it is not bound by the same resource limitations as a hardware transactional memory system is. Further, software is more flexible and easier to evolve than hardware. However, one of the most troublesome drawback of software transactional memory systems is performance. Although, software transactional memory performance has improved over the years, it is still often significantly slower than traditional lock-based and hardware transactional memory solutions.

Meta data structures are necessary in a software transactional memory system in order to manage the state of the ongoing transactions, and Section 5 presents different approaches to do this. For example, conflict detection and resolution in a software transactional memory is done by executing software methods. To track the relation between a transaction and a shared object, the system can either record the objects read or updated by a certain transaction (i.e., track the read- and write-set) or record the transactions that have read or updated a certain object in a *reader set* and a *writer set*, respectively.

Some software transactional memory systems do not track transactional reads of shared objects, a.k.a. *invisible reads*. Thus, such systems cannot detect read-write conflicts. In order to handle the possible inconsistency, three approaches exist: (i) *validation*, i.e., the transaction validates that no other transaction has modified any of the objects in its read-set, (ii), *invalidation*, i.e., track which transactions that read an object and abort them when a transaction opens the object for updates, and (iii), *tolerate inconsistency*, i.e., allow the transactions to execute with an inconsistent state, which in some situations can be tolerable but may result in exceptions or incorrect behavior in other situations (see, e.g., [56]) .

3.5. Synchronization Strategies

One issue when implementing concurrency control in software transactional memory systems is how to handle *synchronization* and *forward progress*. In general, there are two main alternatives: *blocking* and *non-blocking* synchronization. Blocking synchronization is familiar from traditional synchronization primitives such as locks, semaphores, and monitors. Programs written using blocking synchronization cannot guarantee the forward progress of the system, e.g., due to potential problems with deadlocks and priority inversion.

A software transactional memory implementation using non-blocking synchronization can support three levels of forward progress guarantee: (i) *wait-freedom*, (ii) *lock-freedom*, and (iii) *obstruction-freedom*. Wait-freedom [44, 45] is the strongest of the three and guarantees that *all* threads that contend for a set of shared objects make forward progress in a finite amount of time, i.e., system-wide

forward progress is guaranteed. Lock-freedom [32, 20, 75] only guarantees that at least *one* thread of those contending for a set of shared objects makes forward progress in a finite amount of time. Finally, obstruction-freedom [46] is the weakest form of forward progress guarantee. It guarantees that a thread makes forward progress in a finite number of *its own* time steps in the *absence of contention* from other threads for shared objects.

3.6. Contention Management

In order to resolve conflicts between concurrent transactions we often need to abort one of the conflicting transactions. A *contention manager* typically implements one or several *contention management policies* in order to decide which transaction(s) to abort. Several studies have been conducted on contention managers and policies, e.g., [88, 89, 38, 37, 94]. A general conclusion has been that no contention management policy outperforms all other policies in all situations. Examples of contention managers and policies are:

- Greedy [38], which guarantees that each transaction commits within a finite or bounded time.
- Karma [89], which considers the amount processing the conflicting transactions have done so far. The one with the least amount of work done is aborted.
- Polite [48], which uses an exponential back-off strategy to resolve the conflict. When a transaction unsuccessfully has tried to commit a specific number of times, the contention manager aborts the competing transaction(s).
- Polka [89], which backs off for different intervals proportional to the difference in priorities between the transaction and its enemy.
- Time-based [85], which records the start time of a transaction, and in case of a conflict it aborts the transaction(s) with the newest time stamps.
- Timid [88], which always aborts a transaction whenever a conflict occurs.

3.7. Privatization

One problem with software transactional memory systems only supporting weak atomicity (isolation) is the *privatization problem* [95, 91, 1]. The problem occurs when a thread makes some shared object(s) private for, e.g., performance reasons. A typical example is elements in a shared list. A thread may then (inside a transaction) create a private copy of the head pointer to the list. Then, the thread can access the objects in the shared list using its private list head pointer, which is *outside* transactional control. As a result, non-transactional code can possibly access variables that should be protected by transactions but without transactional control. Some solutions to the privatization problem is presented in, e.g., [64, 57].

3.8. Hardware Supported Software Transactional Memory

In order to alleviate some of the performance problems associated with software transactional memory implementations, several researchers have proposed to add some hardware support. One approach is to use hardware signatures to track the read- and write-sets of the transactions, as suggested in, e.g., SigTM [13] and FlexTM [92]. Further, Shriraman et al. [92] suggest four hardware primitives to support high-performance and flexible software transactional memory implementations. Finally, researchers have also proposed to use hardware memory protection mechanisms to implement strongly atomic software transactional memory systems, e.g., [6, 2].

4. Hardware Transactional Memory Proposals

In this section, some of the most important hardware transactional memory implementations are described. They are summarized in Table 1, classified according the how they handle version management and conflict detection. Several of the proposals (shown in italic in Table 1) are described in depth in [56].

Most hardware implementations of transactional memory are not only implemented in hardware. Instead, most of them employ some virtualization technique in order to overcome the limited hardware resources in hardware-only schemes. Therefore, we do not distinguish between hardware-only and virtualized hardware schemes in this section. Looking at Table 1, we can observe that:

1. No proposals exist for the combination of eager version management and lazy conflict detection. This is probably due to semantic problems of how to eagerly update data values, while at the same time postpone detecting conflicts until committing the transaction.
2. *Almost all* recent hardware transactional memory proposals, i.e., from mid 2007 until today, use eager instead of lazy conflict detection.
3. *Most* recent hardware transactional memory proposals favour eager data version management over lazy data version management.

4.1. H&M Transactional Memory

Herlihy and Moss [49] wrote one of the first papers that proposes transactional memory as a way to support lock-free synchronization as efficient as traditional locking techniques, and also as easy to use. They define a transaction as a finite sequence of instruction, executed in a single process, and that satisfies the atomicity and serializability (isolation) properties.

Their transactional memory is a hardware-only implementation that is based on extensions to the cache coherence protocol. Memory is accessed through three instructions that affect the transaction's read- and write-sets: load-transactional, load-transactional-exclusive, and store-transactional. The transactional memory state can

Table 1: A classification of hardware transactional memory proposals, structured the same way as in [70]. The proposals shown in *italic* are thoroughly covered in [56]. Recent proposals, from 2007 and onwards, are marked with an underline.

Conflict detection	**Version management**	
	Lazy	**Eager**
Lazy	*TCC* [41, 39, 40, 67], *Bulk* [14] and BulkSC [15], *TM Semantics* [66, 65], Scalable-TCC [16], FlexTM [92]	No hardware transactional memory proposals.
Eager	*H&M TM* [49], *TLR* [77, 78], *LTM* [4], *VTM* [79], Best Effort HTM [69, 102, 18, 23], FlexTM [92], EazyHTM [101]	*UTM* [4], *LogTM* [70, 71], LogTM-SE [103], ONETM [8], MetaTM [80, 81, 50], TokenTM [11], LiteTM [51], FASTM [60]

be manipulated by three other instructions: commit, abort, and validate (checks if the current transaction has aborted).

The implementation proposal introduces a separate first-level cache that handles all transactional memory accesses. Regular loads and stores are handled by a usual first-level cache. The necessary hardware modifications are limited to the first-level caches and the instructions needed to communicate with them. Transaction commit or abort is thus a local cache operation. Further, the authors propose to rely on the cache coherence protocol to detect conflicts, since the coherence protocol already tracks all memory read and write operations.

4.2. Transactional Coherence and Consistency, TCC

Transactional Coherence and Consistency (TCC) [41, 39, 40, 67, 16] is based on the observation that for well synchronized programs, coherence and consistency are only needed to be maintained at synchronization points. TCC is proposed as a new shared memory model where atomic transactions always are the basic units of work and communication, as well as for memory coherence and consistency. As a result, coherence and consistency is only necessary to maintain at transaction boundaries.

TCC hardware combines multiple writes from the same transaction into a single packet and then sends them to the shared memory as one large atomic block. This bulk communication eliminates the small coherence messages used in conventional coherence protocols. The TCC programming model requires that all accesses to shared data are done within transactions. Thus, TCC can remove the conventional coherence protocol and instead rely only on the read and write-bits for the transaction.

When a transaction is complete in TCC and is ready to commit, the hardware must obtain system-wide permission to commit the transaction's writes. If no conflicts occur, the hardware broadcasts all writes as once. Other nodes in the system snoop on these packages sent at commit time, and can thus detect potential data conflicts between transactions. Each node performs the conflict detection by checking whether any data read by the node has subsequently been written by another transaction. When a conflict is detected, the transaction does a rollback. The snooping version of TCC is modified in [16] to a scalable version based on a directory protocol solution.

The approach in TCC to only maintain coherence at commit points has some interesting implications. Instead of maintaining ordering between individual loads and stores, TCC only needs to maintain ordering between transactions. TCC imposes a sequentially consistent [55] execution of all transactions in the system.

4.3. Bulk and BulkSC

Bulk [14] is the first proposal that use signatures (discussed in Section 3.3) to track the access information, i.e., addresses and read/write information, of a transaction. The access information is hash encoded using Bloom filters [7] into signatures. Thus, the read- and write-sets of a transaction are maintained using a read and a write signature. The signatures are updated on each load and store, and also checked for potential data conflicts for each coherence message that arrives. Two advantages of using signatures are that they support unbounded transactions and nested transactions in a easy way.

Multiple addresses are easily merged into a single signature. However, since the address are hash encoded, the access information is inexact. As a result, there is a risk of so called 'false positives', i.e., a data conflict is detected but for two difference addresses that actually do not have any conflict. This may result in lower performance if the number of false positives is high. Ceze et al. [14] present how the encoding of addresses into signatures can be implemented, as well as the definition and implementation of a number of operations on signatures, e.g., signature intersection and union, empty and membership tests, and signature decoding into sets.

BulkSC [15] supports sequential consistency [55], which presents simple semantics to the programmer. However, implementing sequential consistency with high performance is challenging and may require complex hardware support [33, 82, 35, 34]. The approach in BulkSC is to organize a number of consecutive instructions into chunks of code that appear to execute atomically and in isolation. Chunks are built dynamically by the hardware at run-time, in contrast to transactions that are high-level programming constructs. Memory accesses are allowed to be reordered within each chunk to increase the performance, while a sequential execution order is enforced between chunks. Thus, BulkSC maintains coarse-grain sequential consistency at the hardware level, while still providing a fine-grain (at the memory access level) sequentially consistent model to the programmer.

4.4. Transactional Lock Removal, TLR

Transactional Lock Removal (TLR) [77, 78] is an approach to *dynamically* and *transparently* convert traditional lock-protected critical sections into lock-free transactions. TLR uses Speculative Lock Elision (SLE) [76] as an enabling mechanism in order to create an optimistic transaction. SLE relies on traditional lock acquires in presence of data conflicts. In contrast, TLR relies on a timestamp-based conflict resolution scheme in order to provide serializability and lock-free execution also in cases with data conflicts. The timestamps are based on a local logical clock [54] and the processor identity. Thus, globally unique timestamps are obtained. In case of a conflict, the transaction with the earliest timestamp wins.

The TLR algorithm is executed in four steps.

1. **Calculate timestamp.** Calculate a timestamp for the transaction.
2. **Transaction start.** Initiate TLR mode and remove locks using SLE.
3. **Speculative transactional execution.** The transaction is speculatively executed, updates are buffered locally, conflicts are detected, cache blocks are fetched, etc. If there are insufficient resources or an irreversible operation is encountered, e.g., an I/O operation, TLR acquires the lock before proceeding.
4. **Transaction end.** Commit the transaction; if all cache blocks are in the correct state, e.g., exclusive for writes, update the cache with buffered values, commit transaction register state, service potential waiters, and update local timestamp.

4.5. Unbounded and Large Transactional Memory, UTM & LTM

Unbounded Transactional Memory (UTM) [4] was the first transactional memory proposal supporting unbounded transactions. While many studies at that time argued that most transactions were small (short) and/or occurred infrequently, Ananian et al. argued that a transactional memory system *"should support transactions of arbitrary size and duration"* [4]. In other words, the system shall be able to handle transactions with footprints, i.e., the set of memory locations accessed [100], almost as large as the size of the virtual memory of the system. Further, a transaction can also execute for an arbitrary long time.

UTM stores information about the transaction state in a transaction log, which is a memory-resident data structure called the `xstate`, instead of a processor specific data structure in hardware. As a result, data about a transaction's state is independent of process interrupts, rescheduling, and migration. The `xstate` is shared between all transactions in the system, and contains a log entry for each active transaction in the system. Since the `xstate` is stored in memory, UTM supports unbounded transactions as long as the transaction log fit in virtual memory.

UTM stores new values for a data item in-place in memory, and pointers to blocks with the original values are stored in the `xstate` as a linked list. Each list corresponds to a memory block that has been updated by a transaction. If a potential conflict is detected, the linked list for that block is traversed in order to find out whether other (potentially conflicting) transactions have accessed the block. UTM extends the processor with two new instructions: `XBEGIN pc` and `XEND`. The `pc` argument is the address to an abort handler. If a transaction fails, the processor and memory states are rolled back and the abort handler is executed.

In order to guarantee forward progress when two or more transactions have a conflict, a timestamp-based strategy is used. UTM writes a timestamp into the transaction log the first time a new transaction begins. In case of a conflict, the oldest transaction has priority and the other transactions are aborted.

Large Transactional Memory (LTM) [4] is a simplified version of UTM. UTM requires significant modifications to the processor, cache, and memory system. LTM is a design alternative with less modifications to the hardware, but still with support for large transactions. The major differences between UTM and LTM are as follows:

- LTM only support transaction footprints [100] up to (almost) the size of the physical memory.
- In LTM a transaction must execute and commit within one time slice.
- LTM binds a transaction and its associated state to a particular cache.
- LTM uses lazy version management, while UTM uses eager version management.

4.6. Virtual Transactional Memory, VTM

Hardware transactional memory implementations can provide high performance but their limited hardware resources are a major drawback, which led Rajwar et al. to propose the Virtual Transactional Memory (VTM) [79]. They argue that if transactional memory shall be widely accepted, the programmers must be shielded from platform-specific hardware limitations. For example, limited buffer sizes etc. must not be exposed to the programmer as part of a hardware architecture. If hardware limitations are exposed, it would limit both the flexibility in different hardware implementations and the portability of the code. Further, storing the transaction state in virtual memory makes it a process specific resource, and thus it can migrate to other processors as well as be swapped to disk if necessary. In addition, the transactional memory will also automatically benefit from the protection that the virtual memory system provides.

VTM is a hybrid hardware/software proposal that hides resource exhaustion both in space (e.g, limited buffer space and cache overflow) and time (e.g., context switches, interrupts, scheduling, and process migration). VTM works in two different modes. The first mode is a fast hardware-only

mode for common case transactions that neither exceed the hardware resources nor encounter an interrupt. The second mode is a hardware/software mode that is used for transactions that encounter buffer overflows, context switches, etc. The transactional state in VTM is split into two parts: one part of the state is cached in a processor-local buffers and one part of state, i.e., the overflowed part, resides in the virtual memory of the application.

When a transaction overflows its buffers, the transaction's evicted entries are moved to the Translation Address Data Table (*XADT*) in virtual memory. Similarly, when a time slice expires or an interrupt occurs the transaction's state is saved in the XADT so it can be resumed later. The XADT is consulted each time a transaction issues a load or store that causes a cache miss, to check whether the memory access conflicts with an overflowed address in the XADT. Further, an *XSW* (Transaction Status Word) is defined for each transaction, and since a transaction is associated with exactly one thread the XSW is a part of that thread's state.

4.7. Log-Based Transactional Memory, LogTM

Log-based Transactional Memory (LogTM) [70] is one of the first proposals that focuses on achieving fast commits by using an eager version management approach. The design decision is based on the assumption that commits are more common than aborts. LogTM stores old data values in a per-thread undo log in virtual memory, and updates the memory location with the new value directly. This approach enables fast commits, and the undo log is traversed using software handlers when an abort occurs.

The approach to store the undo log in virtual memory in LogTM has several advantages. Limited hardware support is needed as compared to many other hardware transactional memory proposals, i.e., check-pointing hardware, two hardware pointers to handle the undo log, start address for the software handler, and some counters. Further, LogTM supports unbounded transactions as long as there is space in virtual memory for the undo log. Finally, flushing the undo log is fast since only a change of the hardware pointer to the undo log in memory is needed.

LogTM enables eager conflict detection by using the underlaying hardware cache coherence protocol. One of the contributions of LogTM is the extension of the coherence protocol to handle conflict detection also for blocks that are evicted from the cache due to replacements.

The basic LogTM proposal is extended in [71] to support *nested transactions*. First, the flat closed nesting of transactions in the basic LogTM is extended to support partial aborts in closed nesting allowing inner transactions to abort without aborting the outer transactions. This is implemented using a *stack of activation records* that holds the transaction log for different levels of nesting. Second, open nested transactions is supported, which is used for highly contented resources accessed within transactions. As a result, the parallelism and performance are potentially increased. Third, support is added for calls to lower-level non-transactional code from within a transaction by using *escape actions*, which bypass the transaction version management and conflict detection mechanisms. It is implemented using a per-thread flag that, when set, disables logging and conflict detection.

LogTM is modified in [103] to use signatures to track the read- and write-sets of a transaction, and is then called LogTM Signature Edition, LogTM-SE. Eager conflict detection is done by checking the signatures for each coherence request arriving at the node (cache). By using signatures and a memory log for old values, LogTM-SE decouples transactional memory management from the first-level cache structures. Since both the undo log and the signatures are accessible by software, they are virtualizable, support unbounded transactions, transactions with arbitrary nesting depth (both open and closed nesting), thread migration, and context switches.

4.8. Architectural Semantics for TM

McDonald et al. [66, 65] propose an instruction set architecture (ISA) together with three key mechanisms with well-defined semantics in order to provide a hardware/software interface for transactional memory systems. Well-defined semantics are crucial in order to, e.g., handle composable libraries, implement language or operating system support for transactional memory, and handle I/O and system calls.

The ISA extensions include instructions to start, commit, and abort transactions, instructions to manipulate transaction state registers, manipulate read- and write-sets, and manage violation handlers. The three key mechanisms suggested are:

1. Two-phase transaction commit.
2. Support for software handlers on commit, violation, and abort.
3. Support for nested transactions (both open and closed nesting) with independent roll-back.

Using the three proposed mechanisms above, McDonald et al. [66] show that they provide support for programming languages and operating systems, including support for transparent library calls, system calls, I/O calls, and exceptions within transactions.

4.9. OneTM

Blundell et al. [8] propose an approach to implement unbounded transactional memory, using two synergistic techniques: (i) a *permissions-only cache* that reduces the probability that a transaction overflows, and (ii) *OneTM* which simplifies the implementation of unbounded transactions by allowing only *one* overflowed transaction.

The *permissions-only cache* stores the coherence information (permissions), but no data, for blocks that have been evicted from the processor caches but have been read or written transactionally. As a result, the coherence protocol can detect conflicts also for evicted (overflowed) cache

blocks using only a few hardware bits per evicted block. In the implementation proposal two bits per evicted cache block are used. This corresponds to a 256:1 compression ratio as compared to storing also data, assuming 64-bytes blocks. In other words, the permission-only cache increases the size of bounded transactions by a factor of 256. Thus, the fast case, i.e., bounded transactions that are handled in hardware, will be more common than before.

OneTM is a way to handle unbounded transactions that overflow the permission-only cache, and simplifies the implementation by allowing only *one* overflowed transaction to execute at a time. OneTM relies on the fact that the permission-only cache reduces the number of transactions that actually overflows. OneTM-Serialized and OneTM-Concurrent are two instantiations of OneTM. In OneTM-Serialized, all threads in the same process are stalled when an overflowed transaction executes. In contrast, OneTM-Concurrent allows other non-transactional code as well as non-overflowed transactions to execute concurrently with the single overflowed transaction.

4.10. MetaTM/TxLinux

MetaTM/TxLinux [80, 81, 50] is one of the first efforts to evaluate the impact of large-scale operating system code on the performance of hardware transactional memory systems. MetaTM is the hardware transactional memory implementation that supports TxLinux, a Linux version that is modified to use transactions. MetaTM uses eager version management, eager conflict detection, and supports multiple methods for conflict resolution.

MetaTM uses two instructions, `xbegin` and `xend`, to start a transaction and to commit it, respectively. A third instruction, `xrestart`, restarts a transaction. Nested transactions are not supported in MetaTM. Instead, MetaTM implements stack-based concurrent transactions. The instruction `xpush` suspends the current transaction and stores the transaction state on the stack, and the instruction `xpop` restores the transaction state and continues the execution of a previously stacked transaction.

One of the largest advantages of supporting stack-based transactions is that it simplifies the interaction between I/O (and interrupt handling) and transactions. When an I/O operation (or interrupt) occurs within a transaction, the current transaction is pushed on the stack and restored when the I/O operation is finished.

MetaTM supports the contention management strategies proposed in [89], though adapted to a hardware transactional memory environment. Further, a new policy called SizeMatters is also implemented, which favours transactions with large working sets. In [50] is shown how MetaTM can be combined with a software transactional memory system to form a hybrid transactional memory system.

4.11. Best Effort Hardware Transactional Memory in Rock

Rock [102, 18, 23] is the first commercial processor that supports hardware transactional memory, and it implements a so called *best effort* hardware transactional memory. In order to support transactional memory, two new instructions have been added to the SPARC instruction set — `chkpt` and `commit`.

Best effort hardware transactional memory has a limited number of hardware resources to handle memory transactions. In this case, Rock can handle 32 stores within a transaction. A transaction fails when, e.g., the hardware resources are exhausted, a conflict with another transaction occurs, or a cache line in the read-set is evicted. When a transaction fails, the processor stores the reason in a *checkpoint status register* and transfers the control to the PC-relative offset `fail_pc`, which is specified by the `chkpt` instruction executed when a transaction starts. Depending on the fail reason, the software may retry the transaction, fall back on some software transactional memory implementation, or resort to a software contention manager.

Dice et al. [22] show that Rock's best effort hardware transactional memory can be used to implement and support a variety of transactional memory systems. For example, they have implemented Transactional Lock Elision, which is similar to Speculative Lock Elision [77], HyTM [21], SpHT [58], and PhTM [59]. Finally, an early performance evaluation of the hardware transactional memory in Rock is presented in [23].

4.12. Unbounded Transactional Memory using Tokens, TokenTM

TokenTM [11] is a hardware transactional memory proposal supporting unbounded transactions and is based on the concept of tokens. Each memory block has a number of tokens associated with it. When a transaction needs to read a memory block it acquires *one* token, and when it needs to write to a block it acquires *all* tokens for that block. As a result, a memory block is either (i) not accessed by any transaction, (ii) part of the read-set of one or several transactions, or (iii) part of the write-set of precisely *one* transaction. When a transaction fails to obtain the needed tokens, it detects a conflict and a software contention manager is invoked. The token states are recorded both in per-block metastates and in software-visible per-thread logs.

In order to implement TokenTM efficiently, two novel mechanisms are introduced: (i) *metastate fission/fusion* for efficient modification of tokens by concurrent transactions, and (ii) *fast token release* which enables small transactions to release their tokens in constant time. Using these two mechanisms TokenTM is able to perform fast conflict detection between an arbitrary number of memory blocks, execute small transactions fast, and execute large concurrent transactions without any penalty to non-conflicting transactions. In order for fast token release to work, all blocks associated with a transaction must be present in the processor's first-level cache at the time of transaction commit. Otherwise, the per-thread log must be traversed on commit in order to return all tokens.

4.13. LiteTM

LiteTM [51] is an evolution of TokenTM [11]. Similarly to TokenTM, LiteTM supports unbounded transactions. One of the main contributions of LiteTM is a significantly reduced state overhead (87% lower) for implementing the transactional memory system as compared to TokenTM. This state reduction is accomplished by maintaining only approximate information in hardware about read- and write-sets for the transactions. Exact sharing information is maintained in software. Conflicts are detected in hardware, but the identification of conflicting transactions is done by traversing transactional logs in software.

4.14. Flexible Transactional Memory, FlexTM

Shriraman et al. [92] propose a flexible transactional memory system (FlexTM). FlexTM coordinates four basic hardware mechanisms in order to provide flexibility as well as high performance. The four mechanisms are:

1. *Read and write signatures.* Signatures, first introduced in [14], keep track of the read- and write-sets of a transaction (see Section 3.3).
2. *Per-thread conflict summary tables (CSTs).* CSTs are used to identify and track processor-to-processor conflicts, in contrast to, e.g., conflict detection based on cache lines. Processor-to-processor conflicts are virtualized to thread-to-thread conflict detection.
3. *Programmable data isolation (PDI).* PDI, first introduced in RTM [93], is a lazy version management mechanism, that enables software to perform speculative and incoherent stores to local caches.
4. *Alert-on-update (AOU).* AOU, also first introduced in RTM [93], is a mechanism that allows software to mark specific cache lines. When a coherence request (invalidation) arrives for a marked cache line, a software handler is triggered and executed.

All four mechanisms proposed are accessible from software in order to support virtualization as well as transactions of arbitrary size and length. Further, since the mechanisms are software accessible it is possible to implement a variety of transactional memory systems. For example, Shriraman et al. [92] evaluate and compare both lazy and eager conflict detection in FlexTM. An extended version of FlexTM is presented in one of the papers in this special issue.

4.15. FasTM

FasTM [60] is a hardware transactional memory proposal with eager version management, i.e., new speculative values are stored in-place while old values are stored in a log. A novel feature in FasTM is a new coherence protocol that stores speculative transactional values in the first-level cache while old non-speculative values a kept in the higher levels of the cache hierarchy. As a result, fast abort recovery is enabled as long as transactions do not exhaust the first-level cache resources. Transactional values that overflow the first-level cache are stored in a software managed log, similar to the approach in LogTM [70].

4.16. EazyHTM

EazyHTM [101] is a hardware transactional memory proposal that combines eager conflict *detection* during transactional execution with lazy conflict *resolution* at commit time. In most other proposals are conflict detection and resolution done at the same time. By separating them, higher performance can be obtained since the number of unnecessary aborts is reduced. Transactions are only aborted when some transaction tries to commit, and thus the conflict becomes unavoidable. Further, by detecting conflicts eagerly, the hardware can be simplified by using the existing coherence protocol.

5. Software Transactional Memory Proposals

This section briefly describes some of the most important software transactional memory proposals. Table 2 and Table 3 summarize the main characteristics of the presented proposals. We will not distinguish between pure software transactional memory systems and those that utilize some hardware primitives to enhance the performance.

5.1. Software Transactional Memory, STM

Software Transactional Memory (STM) [90] proposed by Shavit and Touitou is the first implementation of a software transactional memory system. It is word-based with pessimistic concurrency control and a direct update strategy. At the start of a transaction, it identifies and tries to obtain control and ownership of those memory words used in the transaction. If a transaction fails to obtain ownership of a memory location, then it aborts and releases all memory locations it already has acquired. By acquiring memory objects in an increasing order, deadlocks are avoided. Ownership information is stored in a separate meta data structure besides the actual data.

STM uses a direct update policy since it can complete the transaction when it has acquired ownership of all necessary memory locations. It uses a pessimistic concurrency control policy, early conflict detection, and helping for conflict resolution, i.e., if a transaction cannot continue further it aborts and thus helps other transactions complete their execution. Non-blocking synchronization (lock-freedom) is employed in STM. One drawback with STM is that the programmer is required to declare all memory locations accessed within a transaction in advance.

5.2. Word-Based Software Transactional Memory, WSTM

Word-Based Software Transactional Memory (WSTM) [42] is the first software transactional memory that was an integral part of an object-oriented programming language, in this case Java. WSTM supports conflict detection at the word-level, uses optimistic concurrency control with non-blocking synchronization (obstruction-freedom) and late conflict detection, and a deferred update mechanism.

Table 2: A classification of some software transactional memory proposals. The proposals shown in the table are thoroughly covered in [56].

System	Synchronization strategy	Concurrency control	Granularity	Update strategy	Conflict detection	Conflict resolution	Nested transaction support	Isolation
STM [90]	Non-blocking (lock-free)	Pessimistic	Word	Direct	Early	Helping	Not supported	Weak
WSTM [42]	Non-blocking (obstruction-free)	Optimistic	Word	Deferred	Late	Helping	Flattened	Weak
DSTM [48]	Non-blocking (obstruction-free)	Optimistic	Object	Deferred	Early	Contention manager	Flattened	Weak
OSTM [32]	Non-blocking (lock-free)	Optimistic	Object	Deferred	Late	Aborting	Not supported	Weak
ASTM [61, 62]	Non-blocking (obstruction-free)	Optimistic	Object	Deferred	Early or Late	Contention manager	Not supported	Weak
RSTM [63]	Non-blocking (obstruction-free)	Optimistic	Object	Deferred	Early or Late	Contention manager	Flattened	Weak
DSTM2 [47]	Obstruction-free or Blocking	Optimistic	Method	Deferred	Early	Contention manager	Not supported	Weak
McRT-STM [86, 3]	Blocking (lock-based)	Optimistic Rd, Pessimistic Wr	Object, Cache line	Direct	Early Wr-Wr, Late Wr-Rd	Aborting	Closed nested	Weak

In contrast to STM [90], WSTM does not require the programmer to explicitly declare all memory locations accessed within transactions. In addition, WSTM supports nested transactions using flattening. Conflict resolution in WSTM is achieved using helping. The main drawback of WSTM is the high overhead (as most word-based software transactional memory implementations have).

5.3. Dynamic Software Transactional Memory, DSTM

Dynamic Software Transactional Memory (DSTM) [48] overcame the deficiency of previous software transactional memory systems where the transaction size and memory requirements were statically defined in advance. DSTM is designed to handle dynamically sized data structures. It provides C++ and Java APIs for programming dynamic data structures, e.g., lists and trees, for synchronized applications without locks. DSTM employs non-blocking synchronization (obstruction-freedom), optimistic concurrency control with deferred update, early conflict detection at an object-level, and an explicit contention manager for conflict resolution.

Dynamic Software Transactional Memory II (DSTM2) [47], is based on the earlier work on DSTM [48]. DSTM2 is a Java-based library that provides a framework for implementing software transactional memory. It introduces a novel concept, transactional factories, that is used to convert an un-synchronized sequential class into a synchronized one. DSTM2 performs conflict detection at the method level, while DSTM that does it at the object-level. Other differences are that DSTM only supports non-blocking synchronization (obstruction-freedom) but DSTM2 can use non-blocking synchronization or locking, and DSTM supports nested transactions while DSTM2 does not.

5.4. Object-Based Software Transactional Memory, OSTM

Object-Based Software Transactional Memory (OSTM) [32] is the first software transactional memory combining lock-free synchronization and object-based conflict detection granularity.

Each transaction in OSTM has a transaction descriptor, which contains a status field and two linked-lists (one for read-only objects and one for read-write objects). The elements in the read-write list contain an object reference pointing to an object header that points to the real object, and pointers to the original object (old data) and a modifiable copy of the object (new data). Upon a transaction commit, ownership is acquired for all objects in the read-write list. If successful, the object header is updated to point at the new version of the object.

5.5. Adaptive Software Transactional Memory, ASTM

In [61], Marathe et al. compared two software transactional memory implementations, i.e., DSTM [48] and OSTM [32]. They show that, depending on the benchmark, each of the systems has the potential to outperform the other. Further, they provide application characteristics for which each system works best.

Adaptive Software Transactional Memory (ASTM) [62] is based on their observations in [61]. ASTM implements both early and late conflict detection, and can adaptively switch between the two depending on the workload characteristics. The system defaults to early conflict detection but switches to late for transactions that modify few objects but read many objects.

Table 3: A classification of some recent software transactional memory proposals. The proposals shown in the table are not covered in [56].

System	Synchronization strategy	Concurrency control	Granularity	Update strategy	Conflict detection	Conflict resolution	Nested transaction support	Isolation
TL2 [24]	Blocking (lock-based)	Optimistic	Word, Object, Region	Deferred	Early or Late	Aborting	Not supported	Weak
Time-based STM [84, 83, 85]	Non-blocking (obstruction-free)	Optimistic	Word, Object	Deferred	Early	Contention manager	Not supported	Weak
DracoSTM [36]	Blocking (lock-based)	Optimistic	Object	Deferred & Direct	Early or Late	Aborting	Closed nested	Weak
TinySTM [30]	Blocking (lock-based)	Pessimistic	Word	Deferred & Direct	Early	Aborting	Not supported	Weak
SwissTM [27]	Blocking (lock-based)	Optimistic Rd-Wr, Pessimistic Wr-Wr	Word	Deferred	Early Wr-Wr, Late Wr-Rd	Contention manager	Not supported	Weak
Strongly Atomic STM [2]	Blocking (lock-based)	Optimistic Rd-Wr, Pessimistic Wr-Wr	Object	Direct	Early Wr-Wr, Late Wr-Rd	Aborting or Contention manager	Closed nested	Strong
$\mathcal{E}$-STM [31]	Blocking (lock-based)	Pessimistic	Word	Deferred	Early	Aborting or Contention manager	Supported	Weak

5.6. Rochester Software Transactional Memory, RSTM

Rochester Software Transactional Memory (RSTM) [63] is a C++ library implementing an object-based non-blocking software transactional memory system. It supports deferred update, and both early and late conflict detection. Other features of RSTM include a single level of indirection to access data objects, an own memory allocator for use in non-garbage collected languages, and support for several contention management and conflict detection strategies.

RSTM introduces visible and invisible reader lists. An object header has a fixed-size list of transactions that have the object open for reading, i.e., the visible reader list. The transactions in the visible reader list do not need to validate their read data since a conflicting write transaction aborts all transactions in the visible list. If the visible reader list is full, then a new transaction reading an object adds itself to a private invisible reader list. Thus, transactions in the invisible list need to validate their reads.

5.7. McRT-STM

McRT-STM [86] is a software transactional memory system for C++ and Java implemented on top of the McRT [3] run-time system. It employs a direct update strategy in combination with early conflict detection for writes and late conflict detection for reads, and supports conflict detection at both cache line and object level. McRT-STM uses a two-phase locking protocol for synchronization, allowing multiple simultaneous transactional readers of an object but only one transaction can modify an object.

5.8. Transactional Locking II, TL2

Transactional Locking II (TL2) [24], an improvement of Transactional Locking [25, 26], is a software transactional memory system that uses commit time two-phase locking and a global version-clock validation technique. The global clock, i.e., the time stamp, is incremented when a transaction writes to memory and is visible to all transactions. Time stamps have also been used by, e.g., Riegel et al. [83, 85], but their implementation is non-blocking.

TL2 uses deferred update and can select between early or late conflict detection. Each object has a lock and a version number associated with it. Each transaction has a transaction descriptor containing the read and write-sets. Each entry in the read and write-sets has a pointer to the accessed object. A transaction updating an object, adds an entry in the write-set including the new value. A transaction reading an object adds itself to the read-set, and fetches the value either from the write-set (if updated) or from the object. If a conflict is detected, i.e., another transaction has locked the object, the transaction can either delay or abort itself. Upon a commit, a transaction acquires the locks for all the objects in the transaction's write-set, validates its read-set, copies the updated values to the objects, and releases the locks.

5.9. Time-Based Software Transactional Memory

Riegel et al. were the first to present a Time-Based Software Transactional Memory [84, 83, 85], which uses the notion of time to maintain consistency and the order

in which the transactions commit their results. By using a Lazy Snapshot Algorithm (LSA) [83] they can guarantee a consistent view of all objects for all transactional accesses, in contrast to many other software transactional memory systems where consistency only is checked at commit time.

The first implementation of their time-based software transactional memory [84, 83] uses a global counter to maintain the commit order between the transactions. However, a single global counter introduces a potential performance bottleneck. Therefore, Riegel et al. modified their time-based software transactional memory to use a scalable solution [85]. In particular, they showed that the global counter can be replaced by an external or physical clock, and by multiple synchronized physical clocks.

5.10. DracoSTM

DracoSTM [36] is a lock-based C++ library for software transactional memory. DracoSTM is the first software transactional memory implementation that supports both direct and deferred update, and can dynamically switch between the two at run-time. Further, DracoSTM can also dynamically switch between early and late conflict detection. Thus, DracoSTM is a very flexible software transactional memory system. A novel feature in DracoSTM is commit time *invalidation* which, in contrast to validation, can detect priority inversion.

The update strategy is closely coupled to the conflict detection strategy. Early conflict detection is required for the direct update strategy, since it allows only one transaction to update the memory location. Similarly, late conflict detection is performed in conjunction with deferred update. Implementing early conflict detection with deferred update can prevent transactions from committing successfully. Therefore, late conflict detection is coupled with the deferred update strategy.

5.11. TinySTM

TinySTM [30] is a lock-based, time-based, and word-based software transactional memory system. Time-based transactional memories are efficient for read-only transactions, but the read-sets of transactions doing updates must be validated at commit time. Felber et al. [30] suggest that the software transactional memory implementation can be tuned to reduce the validation overhead for transactions with large read-sets.

TinySTM uses several dynamic tuning parameters in order to achieve higher transaction throughput. First, it uses a hash function to map memory locations to locks. TinySTM uses right shifts to control how many contiguous memory locations that will be mapped to the same lock. The second parameter is the number of entries in the lock array. A smaller value maps more memory locations to a single lock, which decreases the size of read-sets and increases the abort rate (due to false sharing). The third is the array size used for hierarchical locking. A larger array size increases the number of atomic operations but decreases the validation overhead and the contention.

5.12. SwissTM

SwissTM [27] is a lock-based and word-based software transactional memory implementation that targets high-performance for large-scale complex transactional workloads. At the same time, it strives to have good performance also for small-scale workloads. SwissTM uses early conflict detection for write-write conflicts, and uses late conflict detection for read-write conflicts.

SwissTM introduces a two-phase contention manager. Read-only and short read-write transactions use a simple and inexpensive abort scheme, where transactions are aborted at the first encountered conflict. More complex long-running transactions dynamically switch to a greedy algorithm, which incurs more overhead but favours long transactions.

5.13. Strongly Atomic STM

The software transactional memory proposal by Abadi et al. [2] provides strong atomicity by detecting conflicts between transactional and non-transactional code using the standard page-level memory protection hardware mechanisms. Potential drawbacks with this approach are overhead for manipulating hardware protection settings, handling access violation faults, and a large conflict detection granularity which may result in false sharing.

Abadi et al. address these issues by using different techniques to reduce the high cost of access violations in off-the-shelf hardware. Conflicts between transactional and non-transactional code are detected by using two different virtual memory mappings for the process heap; one that is used inside a transaction and one that is used in normal execution. The normal software transactional memory, based on the Bartok compiler and runtime system [43], detects conflicts between transactions at an object-level. Further, careful object placement and dynamic code updates of non-transactional code for potential conflicts also reduce the access violation costs.

5.14. Elastic Transactions, $\mathcal{E}$-STM

Elastic transactions [31] is a novel approach to optimize transactional memory for search data structures. An elastic transaction can be divide into several smaller normal transactions depending on different conflict scenarios. The approach is especially well suited for situations where a large part of a data structure is searched (read-only) while updates of the data are localized.

The implementation of elastic transactions, $\mathcal{E}$-STM, is a timestamp-based and lock-based (two-phase locking) software transactional memory, that provides both elastic and normal transactions. Elastic and normal transactions can easily be combined with each other. Upon a conflict, the elastic transaction is divided into two transactions. One implication of this is that the read-set of an elastic transaction only contains one element, i.e., the most recent one.

6. Hybrid Transactional Memory

This section presents some of the most important hybrid transactional memory proposals. A hybrid transactional memory, as classified in this overview, is one that dynamically can switch between hardware and software transactional memory modes during the execution. The presented systems are summarized in Table 4.

6.1. Hybrid Transactional Memory, HyTM

The hybrid transactional memory proposed by Damron et al. (HyTM) [21] relies on a best effort hardware transactional memory implementation. When the hardware resources are exhausted, HyTM switches to a software transactional memory implementation. HyTM has many similarities to Hybrid TM [53], NZTM [99, 98], and the best effort hardware transactional memory in Rock [102, 18, 23]. HyTM provides a word-based interface to the transactional memory, in contrast to Hybrid TM and NZTM that rely on an object-based interface.

HyTM provides a software transactional memory implementation that does not rely on a specific hardware transactional memory support. Instead, HyTM provides the ability to execute transactions using whatever hardware support for transactional memory there are. Further, both hardware managed and software managed transactions can coexist in HyTM. The approach taken to obtain correct interaction between hardware and software transactions is to augment hardware transactions with additional code. Thus, a hardware transaction can check for conflicts with software transactions before it commits.

HyTM assumes similar ISA extensions as many other transactional memory proposals, such as instructions to start, commit, and abort transactions. Damron et al. also assume that some contention control policy is implemented, although HyTM does not require any particular policy to be implemented. HyTM supports nested transactions using flattening.

6.2. Hybrid Transactional Memory, HybridTM

Hybrid transactional memory proposed by Kumar et al. (Hybrid TM) [53], is similar to HyTM [21] and NZTM [99, 98]. Hybrid TM tries to combine the best of two worlds: low overhead and high performance in combination with support for unbounded transactions. When the hardware resources are exhausted or a cache line in the current transaction's working set is evicted, the transaction is aborted and the system resorts to a software transactional memory implementation instead.

Hybrid TM [53] employs two modes of execution for a transaction, a hardware transactional memory mode and a software transactional memory mode. Each transaction can independently choose which mode to use, e.g., based on available hardware resources. The hardware transactional memory implementation requires a buffer to hold transactional data, both new and old values, and some ISA extension, e.g., instructions to start, commit, and abort transactions. However, a standard cache coherence protocol is used for conflict detection for both hardware and software transactions. The software transactional memory part of Hybrid TM is a modified version of the Dynamic Software Transactional Memory (DSTM) [48].

6.3. Phased Transactional Memory, PhTM

Phased transactional memory (PhTM) [59], builds upon the HyTM proposal [21]. The idea behind PhTM is to build a transactional memory system that can switch between different execution modes ("phases") depending on the application behavior. The different modes are executed using different transactional memory implementations. Lev et al. [59] identify five different scenarios and discuss the execution modes that are efficient in each of the scenario. The five execution modes in PhTM are:

- `HARDWARE`. All transactions are executed in hardware.
- `SOFTWARE`. All transactions are executed in software.
- `HYBRID`. Use a hybrid mode, similar to HyTM [21].
- `SEQUENTIAL`. Only one transaction is executed at a time, i.e., no conflict detection is necessary.
- `SEQUENTIAL-NOABORT`. Only one transaction is executed at a time, i.e., no conflict detection is necessary. Further, logging can be elided since transactions do not abort explicitly.

There are several challenges involved when designing a phased transactional memory system. For example, how and when shall the application and system change execution modes. In order to safely switch from one execution mode to another, PhTM does not allow transactions to start in a new mode until all transactions in the previous mode have completed (or aborted). There is a range of policies to use in order to determine when to switch mode and also to which mode. Examples include fixed, scheduled phases of the execution or monitoring various aspects of transaction performance, e.g., commit/abort/restart rates and number of conflicts. Of course, programmer hints or compiler generated directives are also possible approaches.

The prototype implementation of PhTM implements only the `HARDWARE` and `SOFTWARE` execution modes. When compiling an application, the compiler generates two code paths for each atomic block: one for the `HARDWARE` and one for the `SOFTWARE` mode. The compiler used is the same as in HyTM [21], but it is modified to, e.g., allow pure hardware transactions.

6.4. Non-blocking Zero-Indirection Transactional Memory, NZTM

Non-blocking Zero-Indirection Transactional Memory proposed by Tabba et al. (NZTM) [99, 98] relies on a best effort hardware transactional memory implementation. NZTM has many similarities to Hybrid TM [53], HyTM [21], and Rock [102, 18, 23]. When the hardware resources are exhausted, NZTM switches to a software transactional memory implementation.

Table 4: A classification of some hybrid transactional memory system proposals. The proposals shown in *italic* are thoroughly covered in [56]. Recent proposals, from 2007 and onwards, are marked with an underline.

System	Synchronization strategy	Concurrency control	Granularity	Update strategy	Conflict detection	Conflict resolution	Nested transaction support	Isolation
HyTM [21]	Non-blocking (obstruction-free)	Optimistic	Word	Deferred	Late	Contention manager	Flattened	Weak
Hybrid TM [53]	Non-blocking (obstruction-free)	Optimistic and Pessimistic	Object, Cache line	Deferred and Direct	Early/Late	Aborting or Contention manager	Flattened	Weak
PhTM [59]	Non-blocking (obstruction-free)	Optimistic and Pessimistic	Word, Cache line	Deferred or Direct	Early/Late	Contention manager, Aborting, Helping	Sometime supports	Weak and Strong
NZTM [99, 98]	Non-blocking (obstruction-free)	Optimistic	Object, Cache line	Deferred	Late	Contention manager	Not supported	Weak
SigTM [13]	Blocking (lock-based)	Optimistic	Word, Cache line	Deferred	Late	Aborting	Supported	Strong
UFO hybrid TM [6]	Blocking (lock-based)	Pessimistic	Cache line	Direct	Early	Age-based contention manager	Flattened	Strong
SpHT [58]	N/A	Pessimistic	Cache line	Deferred	Early	Contention manager	Open & closed supported	Strong

NZTM provides an object-based transactional memory, which also Hybrid TM [53] does, but in contrast to HyTM [21] that uses a word-based interface. NZTM utilize a best effort hardware transactional memory implementation, i.e., transactions are first executed in hardware. If the hardware transaction fails, it is first retried a number of times in hardware. After a number of unsuccessful attempts in hardware, the transaction switches to a software transactional memory implementation called NZSTM, which is designed to work together with the best effort NZTM.

6.5. Signature-Accelerated Transactional Memory, SigTM

Signature-Accelerated Transactional Memory (SigTM) [13], is a hybrid transactional memory system that uses hardware signatures to track the read and write-sets for penging transactions. Conflict detection is also done in hardware, while all other TM functionality is done in software. One of the main contributions as compared to previous proposals is that it does not modify the underlaying caches and coherence mechanisms. SigTM is also the first hybrid transactions memory system that provides strong isolation between transactional and non-transactional code blocks, without the need for read and write barriers in the non-transactional code.

6.6. UFO hybrid TM

The UFO hybrid TM [6] is a hybrid transactional memory system maintaining strong atomicity that combines three techniques: (i) a *best-effort* hardware transactional memory (BTM), (ii) *fine-grained* hardware memory protection, and (iii) a software transactional memory implementation.

The best-effort hardware transactional memory (BTM) in UFO hybrid TM extends the first-level cache with support for transactional/speculative execution, similarly to the H&M transactional memory [49]. Thus, only transactions that fit in the transactional first-level cache are supported in hardware. The BTM works at the cache line granularity and the speculative execution hardware is exposed to software.

The user-mode fine-grained hardware memory protection scheme assumed in the paper is an enhanced version of iWatcher [104], and associates two *User Fault-On (UFO) bits* with each cache line. These two bits are moved with the memory block throughout the whole memory hierarchy, i.e., in the caches, main memory, and virtual memory (swap file) on disk.

The UFO STM (USTM) is a software transactional memory library for C/C++. Read and write barriers are inserted before accesses to shared data. Thus, USTM performs eager conflict detection. USTM is made strongly atomic by utilizing the UFO bits for transactionally accessed cache lines. Fault-on-write protection is installed at each transactional read access, and both fault-on-read and fault-on-write protection are installed at each transactional write access. Thus, when a non-transactional access results in a conflict, a fault handler is executed.

6.7. Split Hardware Transactions, SpHT

Split Hardware Transactions (SpHT) [58] is an approach to support nested transactions in situations when an underlaying hardware transactional memory implementation does not support nesting. SpHT uses a combination of software and hardware support to divide one large transaction into several smaller segments.

SpHT divides a program-level transaction into a number of smaller segments, and each segment is then executed as a separate hardware transaction. SpHT uses the underlying hardware transactional memory implementation to perform conflict detection and atomic commit. Software support then combines the different hardware transactions into one large atomic block, which corresponds to the program-level transaction. The approach of using split hardware transactions has several advantages, e.g., it overcomes the limitations of "best effort" hardware transactional memory in terms of hardware resources and expressiveness, enables local retry of transactions, provides strong atomicity, and has lower overhead than pure software transactional memory implementations.

7. Concluding remarks

Multicore computers are the main computing platform today and in the future. In order to utilize the performance potential of them, the software also needs to be parallel. However, writing parallel programs is hard and time-consuming. In addition, introducing the necessary synchronizations in a parallel program are difficult and may result in errors that are hard to find, e.g., deadlocks, data races, and priority inversion problems.

Transactional memory has been proposed as a solution to the problems mentioned above as well as enabling software composability. Transactional memory tries to ease the development of parallel programs by providing primitives to declare code sequences as atomic, which can be executed concurrently as long as no data conflicts occur.

This longer than normal introduction to the *JPDC* special issue on transactional memory provides two main contributions in order to support the understanding of the papers included in the special issue. First, it provides a brief description of terminology and concepts for transactional memory in general. Second, it provides an overview of the most important transactional memory proposals up to late 2009. The overview covers both hardware and software, as well as hybrid transactional memory proposals.

It can be noted that there exists a large number of different transactional memory proposals covering many different design alternatives and combinations of policies. Some general trends can be observed. Below, some examples of trends in transactional memory research are noted:

- Most recent hardware transactional memory systems favour eager conflict detection over lazy, as Table 1 shows.
- Eager data version management seems to attend more focus in hardware transactional memory systems than lazy version management does, see Table 1.
- A majority of software transactional memory systems favour optimistic concurrency control over pessimistic.
- Early software transactional memory systems usually employ non-blocking synchronization, as Table 2 shows, while more recent software transactional memory proposals usually employ blocking synchronization, as Table 3 shows.
- Recent software transactional memory proposals usually have a more flexible approach regarding concurrency control, conflict detection, and conflict resolution than older ones, as Tables 2 and 3 show.

Acknowledgment

This special issue had not been possible without the help of a number of people. First of all, I would like to thank Professor Allan Gottlieb, Editor-in-chief of the *Journal of Parallel and Distributed Computing*, for his support of this special issue and encouraging me to write a longer than normal introduction to the special issue. Then, I send a special thanks to Amy Mutale for all of the quality support she has provided throughout the production of this special issue. Next, I would like to thank all the reviewers for their high-quality work during the paper selection process. Finally, I would like to thank Muhammad Nasir for collecting some of the background information for several of the presented software transactional memory systems. The work with compiling this special issue was partly funded by the Industrial Excellence Center EASE - Embedded Applications Software Engineering, (http://ease.cs.lth.se).

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proc. of the 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 63–74, 2008.

[2] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proc. of the 14th ACM Symp. on Principles and Practice of Parallel Programming*, pages 185–196, February 2009.

[3] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proc. of the 2006 Conf. on Programming Language Design and Implementation*, pages 26–37, June 2006.

[4] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proc. of the 11th Int'l Symp. on High-Performance Computer Architecture*, pages 316–327, February 2005.

[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[6] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *Proc. of the 35th Int'l Symp. on Computer Architecture*, pages 115–126, June 2008.

[7] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[8] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proc. of the 34th Int'l Symp. on Computer Architecture*, pages 24–34, 2007.

[9] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[10] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2):17–17, July-December 2006.

[11] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *Proc. of the 35th Int'l Symp. on Computer Architecture*, pages 127–138, June 2008.

[12] J. Bobba, R. Rajwar, and M. Hill. Transactional memory bibliography, 2010. http://www.cs.wisc.edu/trans-memory/biblio/index.html.

[13] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proc. of the 34th Int'l Symp. on Computer Architecture*, pages 69–80, June 2007.

[14] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Proc. of the 33rd Int'l Symp. on Computer Architecture*, pages 227–238, June 2006.

[15] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *Proc. of the 34th Int'l Symp. on Computer Architecture*, pages 278–289, 2007.

[16] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. Cao Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *Proc. of the 13th Int'l Symp. on High-Performance Computer Architecture*, pages 97–108, February 2007.

[17] A. Chang and M. F. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1):28–50, 1988.

[18] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, 2009.

[19] J. Chung, C. Cao Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. Tradeoffs in transactional memory virtualization. In *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 371–381, October 2006.

[20] R. Colvin and B. Dongol. A general technique for proving lock-freedom. *Science of Computer Programming*, 74(3):143–165, 2009.

[21] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 336–346, October 2006.

[22] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the adaptive transactional memory test platform. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, February 2008.

[23] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, March 2009.

[24] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. of the 20th Int'l Symp. Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208, September 2006.

[25] D. Dice and N. Shavit. What really makes transactions faster? In *Proc. of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[26] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proc. of the 2007 Int'l Symp. on Code Generation and Optimization*, pages 21–33, 2007.

[27] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proc. of the 2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 155–165, 2009.

[28] M. Dubois, J. Skeppstedt, and P. Strenström. Essential misses and data traffic in coherence protocols. *J. Parallel Distrib. Comput.*, 29(2):108–125, 1995.

[29] S. Eggers and R. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proc. of the 3rd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 230–242, April 1989.

[30] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 237–246, February 2008.

[31] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Proc. of the 23rd Int'l Symp. on Distributed Computing*, volume 5805 of *LNCS*, pages 93–107, September 2009.

[32] K. Fraser. *Practical lock-freedom*. PhD thesis, Cambridge University Computer Laboratory, February 2004. Also available as Technical Report UCAM-CL-TR-579.

[33] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proc. of the 1991 Int'l Conf. on Parallel Processing*, volume I, Architecture, pages I:355–364, August 1991.

[34] C. Gniady and B. Falsafi. Speculative sequential consistency with little custom storage. In *Proc. of the 11th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 179–188, Sep. 2002.

[35] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proc. of the 26th Int'l Symp. on Computer Architecture*, pages 162–171, May 1999.

[36] J. E. Gottschlich and D. A. Connors. DracoSTM: a practical C++ approach to software transactional memory. In *LCSD '07: Proc. of the 2007 Symp. on Library-Centric Software Design*, pages 52–66, 2007.

[37] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *Proc. of the 19th Int'l Symp. on Distributed Computing*, pages 303–323, Sep 2005.

[38] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proc. of the 24th ACM Symp. on Principles of Distributed Computing*, pages 258–264, Jul 2005.

[39] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6):92–103, Nov-Dec 2004.

[40] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *Proc. of the 11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 1–13, October 2004.

[41] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D.

Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. of the 31st Int'l Symp. on Computer Architecture*, pages 102–113, Jun 2004.

[42] T. Harris and K. Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, October 2003.

[43] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proc. of the 2006 Conf. on Programming Language Design and Implementation*, pages 14–25, June 2006.

[44] M. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proc. of the Seventh ACM Symp. on Principles of Distributed Computing*, pages 276–290, Aug 1988.

[45] M. Herlihy. Wait-free synchronization. *ACM Transactions on Computer Systems*, 13(1):124–149, January 1991.

[46] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of the 23rd Int'l Conf. on Distributed Computing Systems*, pages 522–529, 2003.

[47] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proc. of the 21st ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 253–262, 2006.

[48] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proc. of the 22nd Symp. on Principles of Distributed Computing*, pages 92–101, July 2003.

[49] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Int'l Symp. on Computer Architecture*, pages 289–300, May 1993.

[50] O. S. Hofmann, C. J. Rossbach, and E. Witchel. Maximum benefit from a minimal HTM. In *Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, March 2009.

[51] S. A. R. Jafri, M. Thottethodi, and T. N. Vijaykumar. LiteTM: Reducing transactional state overhead. In *Proc. of the 16th Int'l Symp. on High-Performance Computer Architecture*, pages 81–92, January 2010.

[52] T. F. Knight. An architecture for mostly functional languages. In *LFP '86: Proc. of the ACM Lisp and Functional Programming Conference*, pages 105–112, August 1986.

[53] S. Kumar, M. Chu, C. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proc. of the 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 209–220, March 2006.

[54] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[55] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[56] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[57] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT '09: 4th Workshop on Transactional Computing*, February 2009.

[58] Y. Lev and J.-W. Maessen. Split hardware transactions: True nesting of transactions using best-effort hardware transactional memory. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 197–206, February 2008.

[59] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2007.

[60] M. Lupon, G. Magklis, and A. Gonzalez. FASTM: A log-based hardware transactional memory with fast abort recovery. In *Proc. of the 18th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 293–302, September 2009.

[61] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *Proc. of the 7th Workshop on Languages, Compilers, and Run-time Systems for Scalable Systems*, pages 1–7, October 2004.

[62] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proc. of the 19th Int'l Symp. on Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 354–368, September 2005.

[63] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. In *Proc. of the ACM SIGPLAN Workshop on Transactional Computing*, June 2006.

[64] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *Proc. of the 37th Int'l Conf. on Parallel Processing*, pages 67–74, Sep. 2008.

[65] A. McDonald, B. D. Carlstrom, J. Chung, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Transactional memory: The hardware-software interface. *IEEE Micro, Special Issue on Top Picks from Architecture Conferences*, 27(1):67–76, Jan/Feb 2007.

[66] A. McDonald, J. Chung, D. C. Brian, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *Proc. of the 33rd Int'l Symp. on Computer Architecture*, pages 53–65, June 2006.

[67] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on chip-multiprocessors. In *PACT '05: Proc. of the 14th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 63–74, September 2005.

[68] R. McDougall. Extreme software scaling. *Queue*, 3(7):36–46, 2005.

[69] M. Moir, K. Moore, and D. Nussbaum. The adaptive transactional memory test platform: A tool for experimenting with transactional code for Rock. In *3rd ACM SIGPLAN Workshop on Transactional Computing*. ACM Press, February 2008.

[70] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proc. of the 12th Int'l Symp. on High-Performance Computer Architecture*, pages 254–265, February 2006.

[71] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 359–370, October 2006.

[72] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[73] R. Pagh and F. F. Rodler. Cuckoo hashing. In *ESA 2001: Proc. of the 9th European Symp. on Algorithms*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133, August 2001.

[74] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[75] E. Petrank, M. Musuvathi, and B. Steensgaard. Progress guarantee for parallel programs via bounded lock-freedom. In *Proc. of the 2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 144–154, June 2009.

[76] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proc. of the 34th ACM/IEEE Int'l Symp. on Microarchitecture*, pages 294–305, December 2001.

[77] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. of the 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, October 2002.

[78] R. Rajwar and J. R. Goodman. Transactional execution: Toward reliable, high-performance multithreading. *IEEE Micro*,

23(6):117–125, Nov-Dec 2003.

[79] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proc. of the 32nd Int'l Symp. on Computer Architecture*, pages 494–505, June 2005.

[80] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional memory for an operating system. In *Proc. of the 34th Int'l Symp. on Computer Architecture*, pages 92–103. ACM, June 2007.

[81] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional memory for an operating system. *IEEE Micro*, 28(1):42–51, 2008.

[82] P. Ranganathan, V. S. Pai, and S. V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proc. of the 9th ACM Symp. on Parallel Algorithms and Architectures*, pages 199–210, 1997.

[83] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proc. of the 20th Int'l Symp. on Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 284–298, September 2006.

[84] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *Proc. of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[85] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proc. of the 19th ACM Symp. on Parallelism in Algorithms and Architectures*, pages 221–228, June 2007.

[86] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proc. of the 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 187–197, March 2006.

[87] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *Proc. of the 40th IEEE/ACM Int'l Symp. on Microarchitecture*, pages 123–133, 2007.

[88] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *Proc. of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, Jul 2004.

[89] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proc. of the 24th ACM Symp. on Principles of Distributed Computing*, pages 240–248, July 2005.

[90] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 14th ACM Symp. on Principles of Distributed Computing*, pages 204–213, Aug 1995.

[91] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 78–88, June 2007.

[92] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *Proc. of the 35th Int'l Symp. on Computer Architecture*, pages 139–150, June 2008.

[93] A. Shriraman, M. F. Spear, H. Hossain, V. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proc. of the 34th Int'l Symp. on Computer Architecture*, pages 104–115, June 2007.

[94] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proc. of the 14th ACM Symp. on Principles and Practice of Parallel Programming*, pages 141–150, February 2009.

[95] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *Proc. of the 26th ACM Symp. on Principles of Distributed Computing*, August 2007.

[96] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel & Distributed Technology*, 1(4):58–71, Nov 1993.

[97] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[98] F. Tabba, M. Moir, J. R. Goodman, A. W. Hay, and C. Wang. NZTM: Nonblocking zero-indirection transactional memory. In *Proc. of the 21st Symp. on Parallelism in Algorithms and Architectures*, pages 204–213, August 2009.

[99] F. Tabba, C. Wang, J. R. Goodman, and M. Moir. NZTM: Nonblocking, zero-indirection transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2007.

[100] D. Thiebaut and H. S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, November 1987.

[101] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-lazy hardware transactional memory. In *Proc. of the 42nd Int'l Symp. on Microarchitecture*, pages 145–155, 2009.

[102] M. Tremblay and S. Chaudhry. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. In *Proc. of the 2008 IEEE Int'l Solid-State Circuits Conference*, pages 82–83, February 2008.

[103] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proc. of the 13th Int'l Symp. on High-Performance Computer Architecture*, pages 261–272, February 2007.

[104] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Simple, general architectural support for software debugging. *IEEE Micro*, 24(6):50–56, 2004.