Innovative Research and Applications in Next-Generation High Performance Computing

Qusay F. Hassan
Mansoura University, Egypt

A volume in the Advances in Systems Analysis, Software Engineering, and High Performance Computing (ASASEHPC) Book Series

Information Science REFERENCE
An Imprint of IGI Global

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA, USA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Copyright © 2016 by IGI Global. All rights reserved. No part of this publication may be reproduced, stored or distributed in any form or by any means, electronic or mechanical, including photocopying, without written permission from the publisher. Product or company names used in this set are for identification purposes only. Inclusion of the names of the products or companies does not indicate a claim of ownership by IGI Global of the trademark or registered trademark.

Library of Congress Cataloging-in-Publication Data

Names: Hassan, Qusay F., 1982- editor.
Title: Innovative research and applications in next-generation high performance computing / Qusay F. Hassan, editor.
Description: Hershey : Information Science Reference, 2016. | Includes bibliographical references and index.
Identifiers: LCCN 2016007158| ISBN 9781522502876 (hardcover) | ISBN 9781522502883 (ebook)
Subjects: LCSH: High performance computing.
Classification: LCC QA76.88 .H38 2016 | DDC 004.1/1--dc23 LC record available at https://lccn.loc.gov/2016007158

This book is published in the IGI Global book series Advances in Systems Analysis, Software Engineering, and High Performance Computing (ASASEHPC) (ISSN: 2327-3453; eISSN: 2327-3461)

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

All work contributed to this book is new, previously-unpublished material. The views expressed in this book are those of the authors, but not necessarily of the publisher.

For electronic access to this publication, please contact: eresources@igi-global.com.

Chapter 3
Hardware Transactional Memories: A Survey

Arsalan Shahid
HITEC University, Pakistan

Muhammad Yasir Qadri
University of Essex, UK

Maryam Murad
HITEC University, Taxila Cantt, Pakistan

Nadia N. Qadri
COMSATS Institute of Information Technology, Pakistan

Jameel Ahmed
HITEC University, Pakistan

ABSTRACT

The initiation to have a concept of shared memory in processors has built an opportunity for thread level parallelism. In various applications, synchronization or ordering tools are utilized to have an access to shared data. Traditionally, multithreaded programming models usually suggest a set of low-level primitives, such as locks, to guarantee mutual exclusion. Possession of one or more locks protects access to shared data. But, due to some flaws they become a suboptimal solution. The idea of transactional memory is in research presently as an alternative to locks. Among which, one way is hardware transactional memory. Atomicity is well supported by using transactions in hardware. In this chapter, we have focused on hardware transactional memories and the work done on them so far.

INTRODUCTION

Two or more threads when needed to access some mutual data, then a proper mechanism is required to accurately make those executions. There were some issues in using locks to handle such flaws like mutual exclusion and debugging. Therefore, idea of transactional memory was presented (Guerraoui & Romano, 2014). Basic problem in programming multithreads is to manage shared states (Alessandrini, 2015). A great difficulty is faced when multithreaded programs are created because then the main problem that arises is to synchronize them and made them being able to access the mutual data properly

DOI: 10.4018/978-1-5225-0287-6.ch003

Copyright © 2016, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

(A McDonald, 2009). Programming and computer architecture communities are researching in finding ways to improve parallel techniques of accessing data (Navarro, Hitschfeld-Kahler, & Mateu, 2014). Database systems have successfully exploited parallel hardware for decades. Database is accessed by various operations through transaction even with predictable response.

The idea of transactional memory was first presented by Lomet (Lomet, 1985). He proposed an idea of atomic operations, like the one used in database systems, in programming languages. TM is an attractive feature that access parallel shared data by dealing intellectually with certain issues. TM system is based mainly on three major properties:

- **Atomicity:** States that operations within transactions are all finished successfully, or none of them is executed.
- **Consistency:** Means that each transaction initiates its operation with a consistent view of the shared data and exits the system in a consistent state after completion.
- **Isolations:** Concurrently running transactions are not interrupted by each other (Firoozshahian, 2009).

In TM, programmer defines certain regions that are executed at run-time to detect conflicts automatically. And this is done through read-set and write-set by tracking transactions, and checking reads and writes which are taking place within transactions. For atomicity and isolation, these read sets and write sets are compared with sets of other transactions (A McDonald, 2009).

Transaction Pattern

Transactional memory provides the following (see Table 1) special instructions to access memory instead of the simple load and store.

A transaction's read set is defined as the set of locations loaded using LT, write set as the set of locations loaded using LTX or stored using ST, and data set as the union of the two. The following (see Table 2) instructions are also provided to change the transaction state (Tripathi et al., n.d.).

Listing 1 shows a simple transaction style that shows how the above mentioned primitives can be implemented.

Listing 1. Transaction style

```
Repeat {
    Begin Transaction ();              /* initialize transaction */
    <Read input values>
    Success ← Validate ();             /* test if inputs consistent */
If (success)
    {
        <Generate updates>
        Success ← Commit ();           /* attempt permanent update */
        If (! success)
        Abort ();                      /*terminate if unable to commit */
    }
End Transaction ();                    /* close transaction */
}
Until (success);
```

Adapted from "Transactional Memory part 1".

Table 1. TM instructions

LT	Load-transactional	Loads the value of a shared memory location into a register.
LTX	Load-transactional-exclusive	Same as LT but means that the location is likely to be updated.
ST	Store-transactional	Stores the value from the register to the shared memory location tentatively.

Adapted from Tripathi, Kumar, Faruqui, & Chheda, n. d.

Table 2. Transaction state instructions

Validate	Returns true if the present transaction has not been aborted and false otherwise
Commit	Makes the transaction's changes permanent; succeeds only if no updates to its data set have been made by other transactions and no other transaction has accessed its write set.
Abort	All changes to the transaction's write set are discarded.

Adapted from Tripathi et al., n. d.

First read using LT or LTX from a set of locations then VALIDATE to check the consistency of the values. If it fails then read is performed again.ST is for storing to memory locations from register. In last COMMIT to try to make the transactions changes permanent. If it fails have to start again from read step (Tripathi et al., n. d.).

Researchers have implemented the idea of transactions in various ways. These are usually categorized into three classes:

1. **Software Transactional Memory:** Implementing transactions purely in software at run time (Marathe & Dice, 2014; A. Saha, Chatterjee, Pal, Ghosh, & Chaki, 2015);
2. **Hardware Transactional Memory:** Implements transaction semantics in the hardware (Herlihy, 2014; Leis, Kemper, & Neumann, 2014);
3. **Hybrid Transaction:** Rely on few modifications in underlying hardware (Arcas et al., 2012).

This chapter focuses on HTM systems, because they support transactional mechanisms at minimal overheads and make the implementation details transparent to software. The following section presents the history of transactional memories, their need and major advantages. Further sections present the detailed review of HTM and their mechanism. Moreover, implementations of HTM and advancements in the implementation techniques with time have also been presented. Then some future work is discussed and the final section presents the conclusions.

History of Transactional Memories

The concept of parallel process/thread execution has emerged from past decades. With the trend shifting towards parallel programming, several approaches to synchronize parallel processes have been presented. However, thread synchronization becomes quite difficult when shared memory architectures are under consideration. Many techniques have been presented to overcome this need, i.e., Locks (mutex), Lock-free algorithms (Lockless) etc. One of those technique is to use the concept of transactional memory (TM).

Transactional memory is a technology for addressing concurrent thread synchronization. TM simplifies the parallel programming models by extracting instruction groups to atomic transactions. Concurrent threads operate parallel fashion till they start to modify the same memory chunk. The main advantages for using transactional memories are, 1) easy to use 2) no locks and deadlocks 3) level of performance is increased with the improvement in parallelism. The earliest implementation of transactional memory includes gated store buffer ("Transactional memory," 2015) which was used in Transmeta's Crusoe and Efficeon processors ("Transmeta," 2015).

The following section illustrates a brief overview of Hardware transactional memories (HTM) and their mechanism.

HARDWARE TRANSACTIONAL MEMORIES

Hardware transactional memory can be implemented in three flavors:

1. Full transactional implementation in hardware,
2. HTM synchronized with software, and
3. Hardware extension to software to provide speed-up to parts of software TM.

Hardware implementation relies deeply on memory system to provide the following key capabilities:

- **Tracking:** The memory system has to provide mechanisms to keep track of transactions read and write sets.
- **Buffering:** All results should be buffered inside the memory system.
- **Detecting Conflicts:** The memory system has to detect any potential conflict between any two running transactions in the system (Firoozshahian, 2009).

Thus, an HTM system must be able to recognize memory locations for transactional contacts, manage the read-sets and write-sets of the transactions, detecting and resolving data conflicts, managing architectural register states and committing or aborting transactions. In HTM extensions to the instruction set Tracking read sets and buffering write sets is done using caches and buffers. Coherence messages activate conflict detection. Almost all conventional HTM proposals perform eager conflict detection (Minh, 2008).

Figure 1 shows cache implementation for hardware approach. Caches and shared memory are interconnected through shared bus. They snoop and update contents accordingly.

Each cache has address, value and state to which transactional memory adds a tag (see Figure 2). The address and value pair needs to be consistent across other sets of cache. The basic theme is that any proposed protocol that can detect the conflicts to access can also detect transactional conflicts. Essentially, it is a number of small processor cores, each with some private cache, connected through an interconnect implementing the MESI coherence protocol or some derivative. Keeping in view these basic points, various mechanisms have been presented related to implementing transactions on hardware.

Figure 1. Cache implementation
Adapted from Minh, 2008.

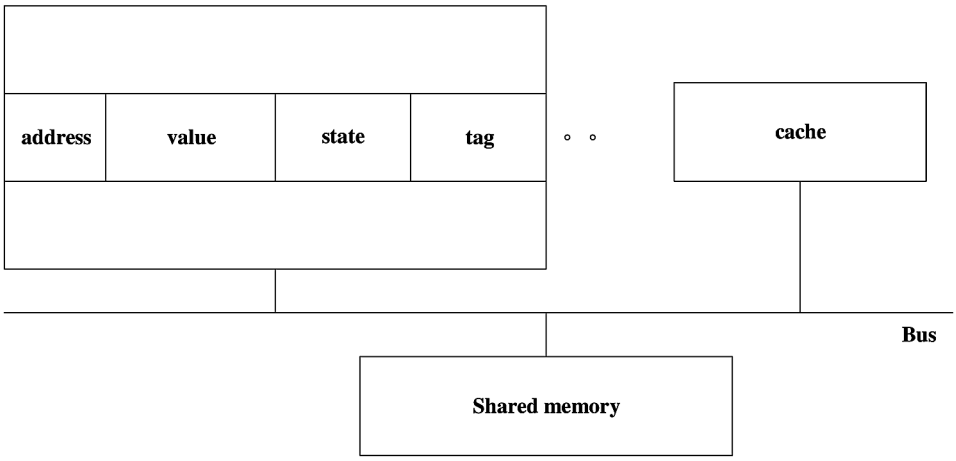
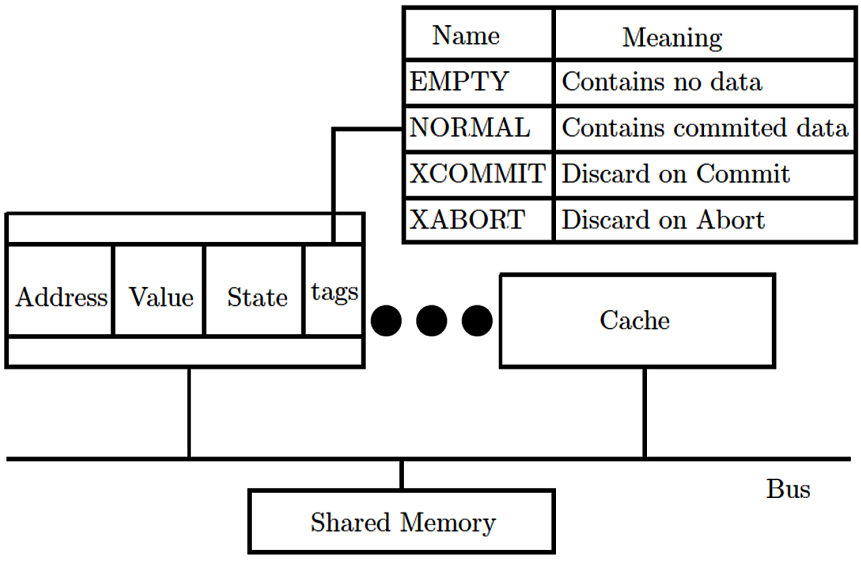


Figure 2. Transactional tag
Adapted from "Architectures for TM," 2009.



HTM Mechanisms

According to researches and various studies related to HTM, the design paradigm can be as in Figure 3. It shows two mechanisms for the implementation of HTM: Versioning and Conflict Detection (see Table 3) ("Architectures for TM," 2009).

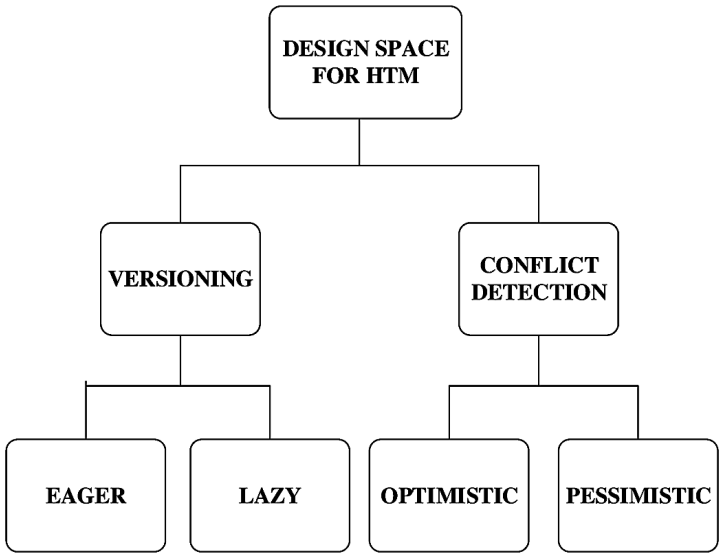
Versioning is to make a transactional code region appear atomic; all its modifications must be stored and kept isolated from other transactions until the commit time. The system does this by implementing a versioning policy. Two versioning paradigms exist: eager and lazy.

Table 3. Implementation approaches

		Versioning	
		Lazy	**Eager**
Conflict Detection	**Optimistic**	Storing updates in write buffer, Detecting conflicts at commit time	Not practical; waiting to update memory until commit time but detecting conflicts at access time guarantees wasted work and provide no advantage
	Pessimistic	Storing updates in a write buffer; detecting conflicts at access time	Updating memory, keeping old values in undo log; detecting conflicts at access time, Log TM

Adapted from "Architectures for TM", 2009.

Figure 3. Design space
Adapted from "Architectures for TM, 2009".



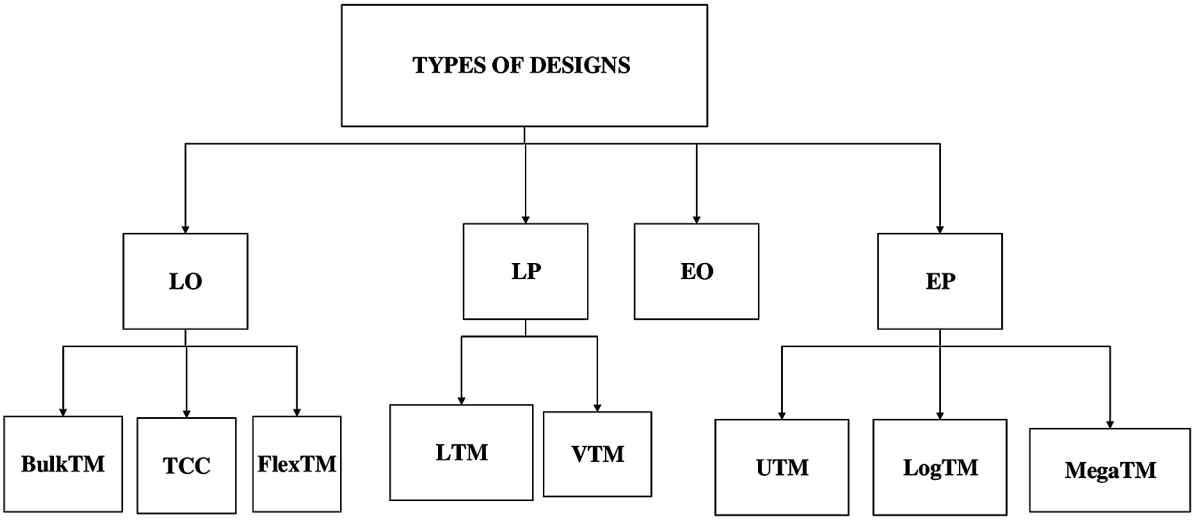
The hardware monitors multiple threads for conflicting memory accesses. Transactions that are not successfully completed are aborted or rolled back. Conflict detection ensures serializability between transactions, conflicts must be detected and resolved. The system detects conflicts by implementing a conflict detection policy, either optimistic or pessimistic. These are the design dimensions that provide choices for HTM designing.

According to some researchers there is also a third type of implementation of HTM, i.e., conflict resolution (Bobba et al., 2007; Yen et al., 2007).

Each TM system needs versioning and conflict detection techniques for implementation. These options give rise to four distinct TM design areas, i.e., Eager-Pessimistic (EA), Eager-Optimistic (EO), Lazy-Pessimistic (LP) and Lazy-Optimistic (LO) (See Figure 4).

These designs provide certain implementation approaches, e.g., Eager-Pessimistic (EP) provides LogTM (K. Moore & others, n. d.; K. E. Moore & Grossman, n. d.), UTM (Akkary, Adl-Tabatabai, Saha, & Rajwar, 2014) and MegaTM (Ramadan et al., 2008); Lazy-Pessimistic (LP) provides LTM and VTM

Figure 4. Design paradigms HTM



(Scott, 2015) design areas and Lazy-Optimistic (LO) is implemented in FlexTM (Dwarkadas, Shriraman, & Scott, 2014), TCC (Hammond et al., 2004) and BulkTM (Ceze, Tuck, Torrellas, & Cascaval, 2006). All of these design approaches are described in following sections.

Hardware Approaches for Transactional Memory

The full hardware implementation idea for transactional memories was introduced by Knight (Knight, 1986) and Moss. Herlihy and Moss (Herlihy & Moss, 1993) gave description of hardware transactional Memory system by describing lock-free data synchronization mechanism. They showed that by making few changes in cache, those transactions that are completed without using context switching can be well supported. They proposed a protocol that was capable to detect conflicts related to access. Two versions of transactional memory were implemented based on protocols, i.e., Goodman's protocol and chicken directory protocol. Their results showed that transactional memory is more competitive than other lock-based techniques for simple benchmark applications, i.e., counting, producer/consumer and doubly-linked list benchmarks. Moreover, the approach was limited to assumptions like short transactions duration and small data sets. After this, transactional memory has been worked out in various ways.

Rajwar et al. (R. Rajwar & Goodman, 2002) remarked on Herlihy and Moss' proposal that it is not prime because it has certain restrictions as it cannot be accomplished without a well programmer and some extensions to protocols. Lev et al. (Lev & Maessen, 2008) further added Herlihy and Moss work is not that much robust; it can only work for certain transactions with size limitations. They also highlighted the point that it is based on architecture and is not that suitable. Some of these problems were then tried to overcome in Rajwar et al.'s work that was on Virtualizing Transactional Memory (Ravi Rajwar, Herlihy, & Lai, 2005).

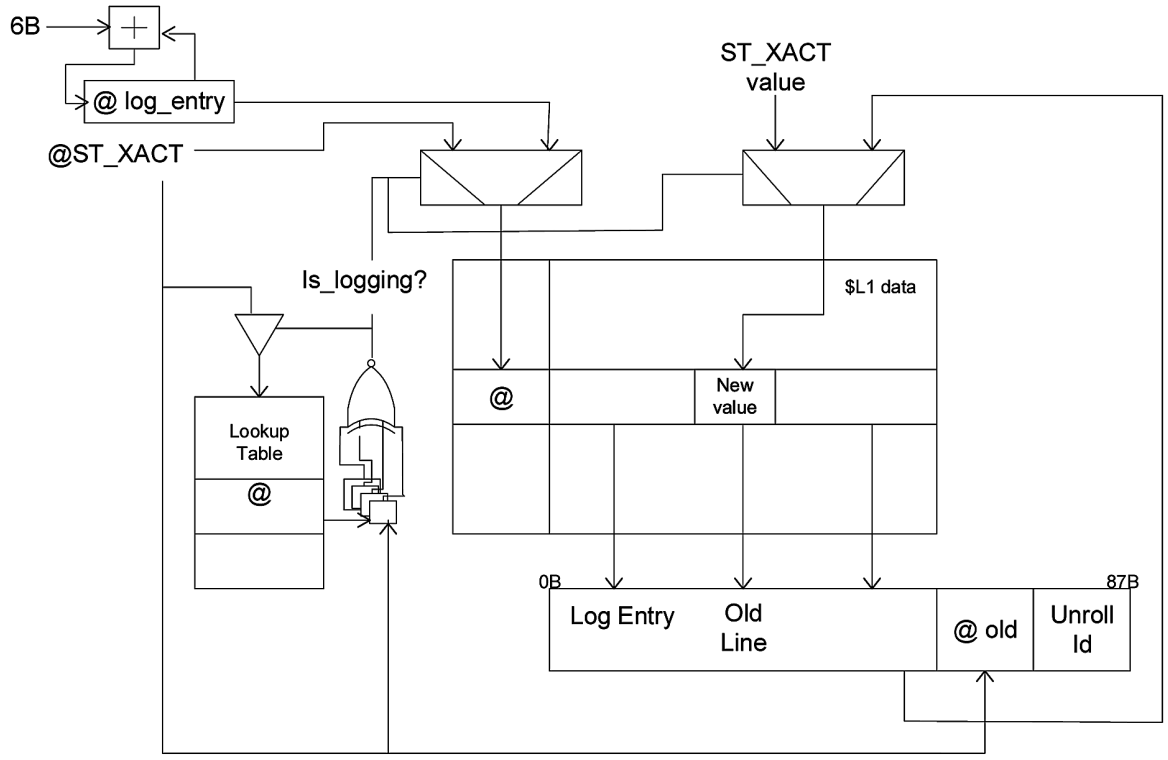
LogTM (K. E. Moore & Grossman, n.d.) Is an eager version management HTM that stores old transactional values in a private software log. This eliminates commit actions because all the changes introduced by transactions are in place, but requires a software handler to restore transactional state in

case of abort. LogTM refinements (Moravan et al., 2006; Moss & Hosking, 2006) minimize the hardware resources needed to track transactional state. In its first incarnation, LogTM used RW bits in the cache and in the directory to track the access summary, using line granularity. Later, LogTM-SE (Yen et al., 2007) introduced signatures to decouple RW bits from caches.

A LogTM design is able to read the old line and write the new value in the same cache access. The old line is momentarily stored in a latch, which also contains the modified memory location. Assuming a 64 byte cache line and a 32-bit processor, an undo log entry consists of 64 bytes to keep the old cache line, 26 bits to track its address and 6 bits that identifies it as an undo entry. Notice that a log entry requires 68 bytes, distributed in two different cache lines. This means that log entries are not aligned at cache line boundaries. Moreover, the system has to keep a log pointer that tracks where the entry is located in the software log. This pointer is an address that is incremented by the entry size. Hence, once the old data is placed in the latch, the system can store the undo entry in the log, moving the latch values plus the identifier to the memory hierarchy. These two movements do not need to check the RW bits of other transactions, because they update a memory section that is private to the thread.

Logging has a non-negligible cost, because it requires several movements to memory. For that reason, two different improvements can be implemented to reduce the time spent logging. First, a table can be used to track those addresses that have been written before inside this transaction. These lines are present in the log, so there is no necessity to replicate them. If the address is not in the table, a new undo entry is stored in the log, although the log already contains it. That is not a problem, because, if an abort occurs, the log is restored in reverse order. Figure 5 shows a diagram of the logging process.

Figure 5. Log process in LogTM
Adapted from "Hardware Approaches for Transactional Memory".



Nested LogTM supports nesting by segmenting the log into a stack of activation records and modestly replicating R/W bits. Open nesting is done by allowing a committing open transaction to release isolation and optionally save commit and compensating actions on the log, and non-transactional escape actions, also with commit and compensating actions (Moravan et al., 2006).

Eager Version Management Transactional Memory (EVM) stores new values in the memory hierarchy and the old values in the buffer. EVM operates identically to LogTM, but it accelerates its version management mechanism with the Transactional Buffer. Figure 6 overviews different operations that are responsible for EVM by using Transactional Buffer. EVM keeps old values in the buffer; whereas, new values are in the memory hierarchy. A transactional store searches if a location is present in the Transactional Buffer. If it is present, no action is performed in the buffer; otherwise the old value is introduced in the buffer, with its own memory address. A commit pushes the valid bits of all the entries of the Transactional Buffer, whereas an abort stores the values from the buffer to the memory hierarchy.

Lazy Version Management Transactional Memory (LVM) uses the Transactional Buffer to store speculative transactional values, keeping old values in the memory hierarchy. LVM operates with the other mechanisms of the LogTM, such as conflict detection or conflict resolution, but the version management approach is slightly different, modifying the commit and abort phase, like Rock (Moir, Moore, & Nussbaum, 2008) or BulkTM (Moir et al., 2008).

Figure 6. EVM using transactional buffer
Adapted from "Hardware Approaches for Transactional Memory".

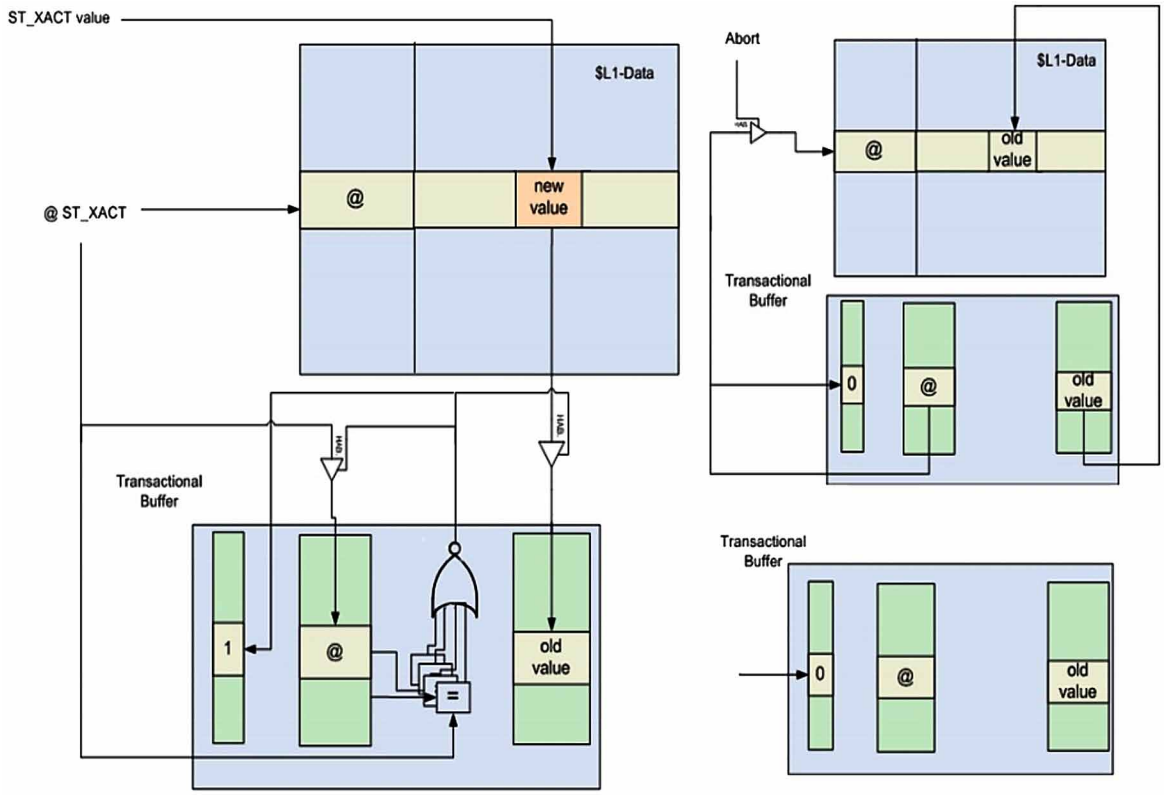
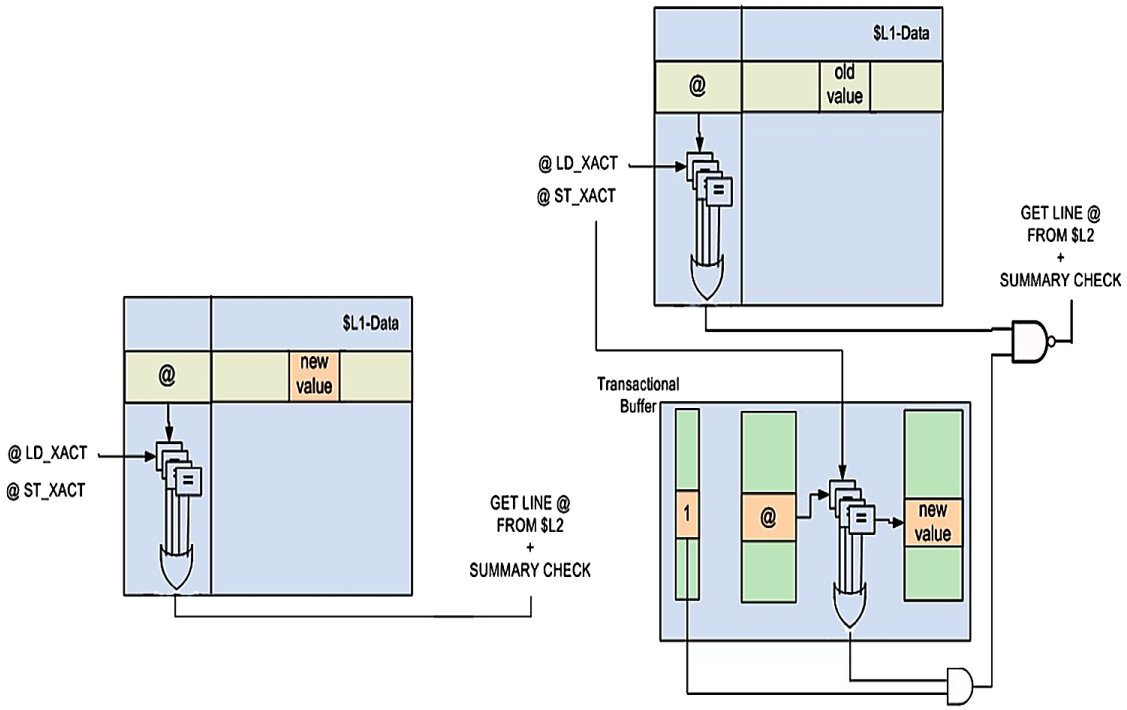


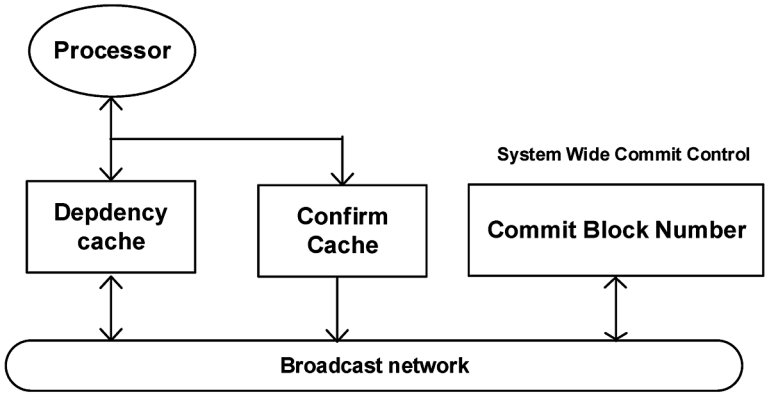
Figure 7. Eager conflict detection in EVM and LVM
Adapted from "Hardware Approaches for Transactional Memory".



Lazy Conflict Detection Transactional Memory (LCD) emulates a Transactional memory Coherence and Consistency (TCC) environment (Hammond et al., 2004), with two main differences. First, TCC doesn't allow interference between non-transactional and transactional code. Second, TCC uses a write-through policy, where a shared L2 cache contains the consistent state of the committed transactions, so the L1 cache just keeps the speculative state.

LVM keeps new values in the buffer whereas old values are in the memory hierarchy. A transactional load looks at the buffer if the line is present, and if it is it returns the value stored in the buffer. If not, it returns the value from the memory hierarchy. A transactional store searches if a location is present in the Transactional Buffer. If it is present, the value is updated in the buffer; otherwise a new entry is added in the buffer. An abort pushes the valid bits of all the entries of the Transactional Buffer, whereas a commit moves the values from the buffer to the memory hierarchy. However, there is an exception where a transactional store modifies both the memory hierarchy and the Transactional Buffer. System calls usually modify memory locations, and these updates must be done in the memory hierarchy. Otherwise, the Operation System does not know where they are stored. Some of these system calls are executed inside a transaction, and must be done atomically. Fortunately, most of these calls (like malloc or free) don't need to be reversed or allow some compensation code if a transaction aborts. Each store from a system call is marked as privileged store, which permits the hardware to send it to the memory. Figure 7 resumes how EVM and LVM perform eager conflict detection. Basically, they keep the coherency using a MESI transactional protocol, checking the access summary if the line is not present in the cache or if is shared. However, LVM doesn't request the line to the L2 if it is present in the Transactional Buffer, so it reduces the track for evicted lines. EVM and LVM improve LogTM's version management, but they keep its conflict detection philosophy.

Figure 8. Knight's hardware organization
Adapted from Herlihy & Moss, 1993.



A modified version of coherence protocol was presented in LOGTM (K. Moore & others, n.d.). It allowed transactional phase to escape phase. Whereas, TCC version management keeps new data in a speculative cache until commit, when they are written through to a shared L2 cache, LogTM can operate with writeback caches and generates no traffic at commit. In TCC's lazy conflict detection, other transactions learn about transaction T's conflicting store when T commits, not earlier when the store is executed (Tripathi et al., n.d.).

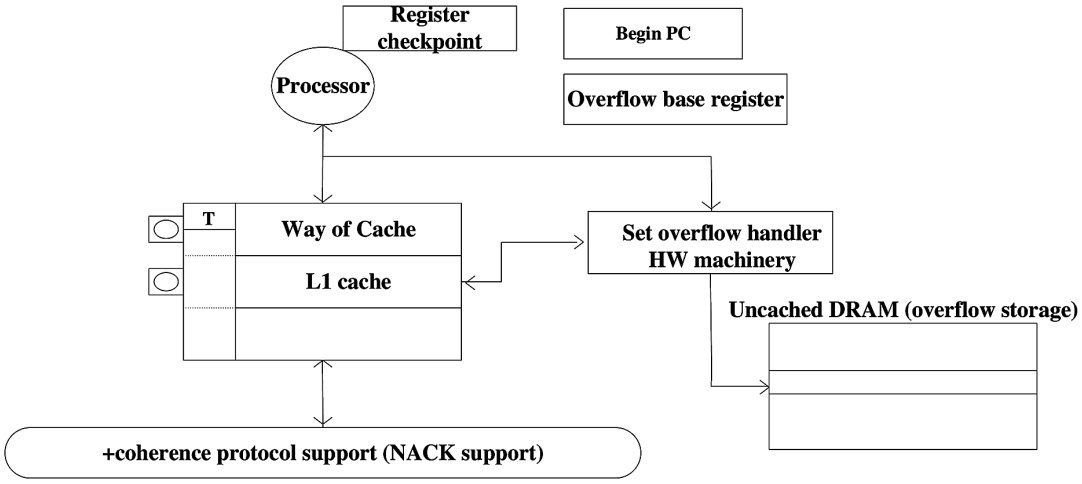
In Knight's proposal, TCC requires the program to be divided in a series of transactions—either a single threaded program is divided into a series of transactions, or a programmer may specify explicit transactions in a multithreaded environment (see Figure 8) (Herlihy & Moss, 1993).

Like LogTM, LTM keeps new data in cache when it can. However, when a transaction overflows a set in the cache, LTM stores new values in an uncatchable memory hash table (see Figure 9). On commit, LTM copies overflowed data to its new location. In contrast, LogTM allows both old and new versions to be cached (often generating no memory traffic) and never copies data on commit. Whereas, an LTM processor must search a table in uncatchable memory on an incoming request to any set that has overflowed a block during the transaction, a LogTM processor needs to check only local state allowing it to respond immediately to a directory request (Tripathi et al., n.d.).

Ananian et al. (Akkary et al., 2014) proposed an extension where evicted transactional lines in an HTM would be moved by hardware into a dedicated area in local memory without aborting. Their HTM system, called Large Transactional Memory allocates a special uncashed region in local memory to buffer transaction state that spills from the cache. This region is maintained as a hash table. Each cache set has an associated overflow bit (O bit). This bit is set when a transactional cache line (tracked using a T bit) is evicted and moved into the hash table. During this process, the processor responds to incoming snoops with a negative acknowledgment (NACK). The processor does not service incoming snoops until it has checked both the cache and the memory hash table for a conflict (Herlihy & Moss, 1993).

UTM version management stores new values in place and old values in a log. UTM's log is larger, however, because it contains blocks that are targets of both loads and stores, whereas LogTM's log only contains blocks targeted by stores. UTM uses this extra log state to provide more complete virtualization of conflict detection, allowing transactions to survive paging, context switching and thread migration. UTM's conflict detection must, however, walk the log on certain coherence requests, and clean up log

Figure 9. LTM design to overflow cache lines into DRAM
Adapted from (Herlihy & Moss, 1993.



state on commit, while LogTM uses a simple directory protocol extension (that does not even know the location of the log) and uses lazy cleanup to optimize commits (Tripathi et al., n. d.). UTM is an idealized design for HTM that requires significant changes to both the processor and the memory subsystem of current computer architecture. UTM adds two new instructions to a processor's instruction:

- **XBEGIN PC:** Begin a new transaction. The pc argument to XBEGIN specifies the address of an abort handler (e.g., using a PC-relative offset). If at any time during the execution of a transaction the hardware determines that the transaction must fail, it immediately rolls back the processor and memory state to what it was when XBEGIN was executed, then jumps to pc to execute the abort handler.
- **XEND:** End the current transaction. If XEND completes, then the transaction is committed, and all of its operations appear to be atomic with respect to any other transaction (Akkary et al., 2014).

Another HTM design that has improved hardware to handle infinite sized transactions is TM (Moravan et al., 2006). For unbounded transactions it simply focuses on hardware to permit those transactions and to permit unbounded transaction sizes, but it limits the implementation to single unbounded transaction at specific time interval. This is then handled by assigning tokens to blocks of memories that enables them to execute without conflicts. In TM (Dwarkadas et al., 2014), transactions that were not overflowed were not affected by those transactions that were overflowed. Moreover, some schemes were also proposed for hardware implementations to aid software's. Like in METATM (Ramadan et al., 2008), that supports transactional operating system (TXLINUX (Rossbach et al., 2007)) by providing architectural support to run. Another scheme Flex-TM (Dwarkadas et al., 2014) presents a transactional hardware. It decoupled hardware into four components, access tracing, conflict notification, data versioning support and conflict tracking. These properly defined sections make non transactional implementation easy (Navarro et al., 2014).

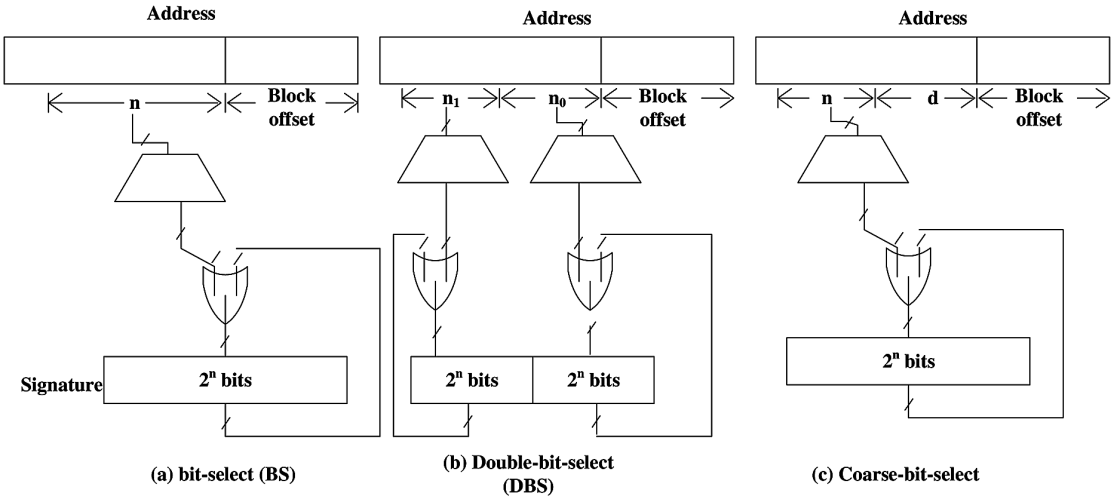
LogTM-SE (A. Saha et al., 2015), a log-based HTM system that uses eager version management. LogTM-SE decouples transactions from caches using signatures (large bit tables) to store transactional accesses, which permits the elimination of RW bits in the L1 and the sticky states in the L2. Lots of addresses are stored in the signature using hash functions, which might introduce false positives. LogTM-SE requires a signature per nesting level, and they are checked in parallel to detect conflicts. If signatures are not available, they are virtualized.

Saha et al. (B. Saha, Adl-Tabatabai, Hudson, Minh, & Hertzberg, 2006, p. -) proposed a scheme hardware accelerated STM (HASTM).It was a suggested hardware support to reduce the overhead of software transactional memory instrumentation. Performance and even flexibility has been the main focuses in hardware transactional memory related work. To improve the performance and to make things flexible remained the main theme. A research ("Architectures for TM, 2009) compared two different approaches for hardware transactional memories. They compared Eager-Eager and Lazy-Lazy to observe their respective energy utilization and performance. And they concluded that eager approach requires improving their energy consumption.

Figure 10 shows the typical signature operations in LogTM-SE. LogTM-SE supports operations on signatures similar to Bulk: adding an address to a signature, checking whether an address aliases with a signature, and clearing a signature's "O" set. Unlike Bulk, LogTM-SE continues to use the existing cache coherence protocol for conflict detection, rather than relying on separate global broadcasts of signatures or tokens. Conflicts are detected by comparing the address in an incoming coherence request against the local signatures.

The Rock (Dice, Lev, Moir, Nussbaum, & Olszewski, 2009) processor is assumed as the first to include transactional hardware support. A gated buffer is used to store the alterations in transactions and when the store is filled, the system gives an indication through interrupts and software then takes corresponding action.

Figure 10. LogTM-SE signatures
Adapted from Yen et al., 2007.



TokenTM (Bobba, Goyal, Hill, Swift, & Wood, 2008) used the idea of tokens to detect the issues related to memory blocks and then implementing them again by a different mechanism. Those mechanisms were either fission or fast token process.to correctly track conflicts on an unrestrained number of memory blocks and implements them with new mechanisms, including metastatic fission/fusion and fast token release. By TokenTM small sized transactions were executed in less time and large transactions were dealt parallel without any conflict. Thus, context switching and paging was dealt in a good manner.

Implementation of speculative lock elision (Ravi Rajwar & Goodman, 2001) in hardware was also proposed to execute lock regions as transactions. It enabled users to use frequent locks to some extent. Speculative lock elision or SLR was extended to Transactional lock removal TLR (Hammond et al., 2004) that used the idea of time stamps to resolve arising conflicts in transactions. Afterwards, HTM system focused SLR by limiting its size of transactions.

To avoid any modifications to the data cache for tracking read sets and write sets, they considered an HTM design with 32 KB 4 way set associative data cache with 64 byte cache lines and found that such an HTM could frequently support transactions up to 30,000 instructions that operate on hundreds of cache line, that was fairly large compared to common perceptions (Minh, 2008).

Zilles and Rajwar (Zilles & Rajwar, 2007) studied the effectiveness of data cache for tracking read sets and write sets. They considered an HTM design with a 32 KB four-way associative set data cache with 64 bit cache lines and found that such system could repeatedly support transactions for about 30,000 instructions that implements on several of cache lines that were equally large compared to common observations. However, in this kind of systems, a transaction must be aborted whenever an overflow occurs.

Khan et al. (Khan et al., 2008) integrate transaction management with the object translation buffer used in processor with an object-based model implementation instead of orthodox cache line model. The Azul (Click, 2009) HTM system uses data cache to track accesses and buffer transactional state. However, it relies on software to perform register recovery. KILO TM (Fung, Singh, Brownsword, & Aamodt, 2011) scheme presents hardware transaction design for GPU's that scales many parallel transactions. It is not dependent on cache coherent hardware. It utilizes conflict detection at word level based on values in order to reduce the storage overhead on chip.

Another proposal by Epifanio et al. (Gaona, Titos-Gil, Acacio, & Fernández, 2012) is implemented on top of a hardware transactional memory system in which eager conflicts manages and detects transactions mechanism. It also serializes conflicting transactions in a dynamic way. Especially when there is a conflict only single conflict is allowed to work out further. In Lazy hardware transactional memory (Negi, Titos-Gil, Acacio, Garcia, & Stenstrom, 2012), which is more competent way, available parallel part is utilized instead of eager one. They computed the issues related to the problem and developed a scheme named π-TM that is early conflict detection and a modified lazy conflict resolution design. This design focuses on how to make extensions to existing directory based protocols and how to utilize the available information to attain the proper parallelism concept.

Tor M. et al. (Fung, Singh, Brownsword, & Aamodt, 2012) explored how transactional memory concept can support GPU systems. The main challenge was to make it easy to write corresponding programming model as it requires very well defined synchronization to cope with several threads. The proposed TM system optimally runs transactions in parallel sequence. In this system, if data race occurs between any transactions, the system restarts any of the transaction in order to insure atomic appearance of transactions. In Fact, eager implementations are relatively simpler to handle in cache coherence architectures as compared to lazy scheme. In view of this fact related to HTM techniques, it was claimed that if stores are properly handled, more challenging design alternatives can be proposed (Titos-Gil, Negi, Acacio, Garcia, & Stenstrom, 2013).

RECENT IMPLEMENTATIONS OF HTM

IBM launched BlueGene/Q processors for Livermore National Labs. It was provided in 2011 and was functionally deployed in 2012. It provides power to the 20 petaflops Sequoia supercomputer ("IBM Sequoia," 2015). They are first commercial processors providing hardware support for transactional memory. The transactional memory has been configured in two modes; one is an unordered and single-version mode, where a write from one transaction causes a conflict with any transactions reading the same memory address. The second mode is for speculative multithreading, providing an ordered, multi-versioned transactional memory. IBM Blue Gene/Q multicore will be 64-bit PowerPC-based system-on-chip based on IBM's four-way multithreaded PowerPC A2 design with 1.47 billion transistor chip TM that will appear in 32 MB level 2 cache 18 cores 16 for running actual computations 1 for the operating system 1 to improve chip reliability.

Intel Haswell launched its Haswell architecture corresponding to transactional memory in 2013. It was supporting hardware for memory transactions. These transactions were named as Transactional Synchronization Extensions (TSX) by Intel. They will be extended in to two parts, Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). HLE will enable to convert lock based programs into transactional related programs and that will be in line with present processor. (RTM) will be a full implementation of transactions on memory.

HTM, despite of many limitations, allows for efficient concurrent and atomic operations; which is considered to be very significant in context of databases. Viktor et al. (Leis et al., 2014) presented a mechanism to exploit HTM in main memory databases. Moreover, they evaluated transaction management scheme to transforms a large database transaction into a sequence of elementary and single tuple HTM transactions.

Most recently, Transactional memory support has been added to IBM POWER8 Processors by IBM (Le et al., 2015). The POWER8 transactional memory facility provides a robust capability to execute transactions that can survive interrupts. Moreover, it provides the first implementation of hardware transactional memory that is supported directly by the Power ISA. The architecture also provides Rollback-Only Transactions to permit transactional memory to be used to execute single-thread code.

AMD proposed Advanced Synchronization Facility (ASF), i.e., the x86-64 instruction set architecture that would add hardware transactional memory support ("Advanced Synchronization Facility," 2015). Up till now, none of the released microprocessors included this support. However, it is expected to be launched soon in the upcoming versions.

CONCLUSION

Shared memory architectures have emerged as mainstream processor architectures. Trend shifting towards shared memory systems have motivated researchers for thread level parallelism. Many people have been using locks for synchronization and thread ordering. Transactional memory is an approach to address the same problem as an alternative to locks as it supports atomicity. The design area related to hardware transactional memory systems has been studied widely by researchers. To achieve true optimism is dealt in various policies, ordinary lazy designs are a bit suspicious. Eager designs are pessimistic in cases where they detect conflicts. This chapter presents a survey on researches so far done and some of the proposed schemes related to hardware transactional memories. Moreover, some of the recent implementations of HTM have also been presented.

REFERENCES

Advanced Synchronization Facility. (n. d.). *Wikipedia, the free encyclopedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Advanced_Synchronization_Facility&oldid=680636378

Akkary, H. H., Adl-Tabatabai, A.-R., Saha, B., & Rajwar, R. (2014). *Unbounded transactional memory systems*. Google Patents.

Alessandrini, V. (2015). *Shared Memory Application Programming: Concepts and strategies in multicore application programming*. Morgan Kaufmann.

Arcas, O., Kirchhofer, P., Sönmez, N., Schindewolf, M., Unsal, O. S., Karl, W., & Cristal, A. (2012). A low-overhead profiling and visualization framework for Hybrid Transactional Memory. *Proceedings of the 2012 IEEE 20th Annual International Symposium on In Field-Programmable Custom Computing Machines* (FCCM) (pp. 1–8). IEEE.

Bobba, J., Goyal, N., Hill, M. D., Swift, M. M., & Wood, D. A. (2008). Tokentm: Efficient execution of large transactions with hardware transactional memory. ACM SIGARCH Computer Architecture News, 36(3), pp. 127–138. IEEE Computer Society.

Bobba, J., Moore, K. E., Volos, H., Yen, L., Hill, M. D., Swift, M. M., & Wood, D. A. (2007). Performance pathologies in hardware transactional memory. In *ACM SIGARCH Computer Architecture News* (Vol. 35, pp. 81–91). ACM.

Ceze, L., Tuck, J., Torrellas, J., & Cascaval, C. (2006). Bulk disambiguation of speculative threads in multiprocessors. ACM SIGARCH Computer Architecture News, 34(2), 227–238). ACM. doi:doi:10.1109/ISCA.2006.13 doi:10.1109/ISCA.2006.13

Click, C. (2009). Azul's experiences with hardware transactional memory. *Proceedings of theTransactional Memory Workshop*.

Dice, D., Lev, Y., Moir, M., Nussbaum, D., & Olszewski, M. (2009). *Early experience with a commercial hardware transactional memory implementation*. Sun Microsystems. doi:10.1145/1508244.1508263

Dwarkadas, S., Shriraman, A., & Scott, M. (2014). *Mechanism to support flexible decoupled transactional memory*. Google Patents.

Firoozshahian, A. (2009). *Smart memories: A reconfigurable memory system architecture*. ProQuest.

Fung, W. W., Singh, I., Brownsword, A., & Aamodt, T. M. (2011). Hardware transactional memory for GPU architectures.*Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 296–307). ACM.

Fung, W. W., Singh, I., Brownsword, A., & Aamodt, T. M. (2012). *Kilo tm: Hardware transactional memory for gpu architectures* (Vol. 3, pp. 7–16). IEEE Micro.

Gaona, E., Titos-Gil, R., Acacio, M. E., & Fernández, J. (2012). Dynamic Serialization: Improving Energy Consumption in Eager-Eager Hardware Transactional Memory Systems. *Proceedings of the 2012 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (pp. 221–228). IEEE.

Guerraoui, R., & Romano, P. (2014). *Transactional Memory. Foundations, Algorithms, Tools, and Applications, LCNS* (Vol. 8913). Springer.

Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., & Olukotun, K. (2004). Transactional memory coherence and consistency. In *ACM SIGARCH Computer Architecture News* (Vol. 32, p. 102). IEEE Computer Society.

Hardware Approaches for Transactional Memory. (n. d.). Retrieved from http://arco.e.ac.upc.edu/wiki/images/e/e3/Mlupon_msc.pdf

Herlihy, M. (2014). Fun with hardware transactional memory.*Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (pp. 575–575). ACM. doi:10.1145/2588555.2602132

Herlihy, M., & Moss, J. E. B. (1993). *Transactional memory: Architectural support for lock-free data structures* (Vol. 21). ACM. doi:10.1145/165123.165164

IEEE Transactional memory. (n. d.). *Wikipedia, the free encyclopedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Transactional_memory

Khan, B., Horsnell, M., Rogers, I., Luján, M., Dinn, A., & Watson, I. (2008). A first insight into object-aware hardware transactional memory.*Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures* (pp. 107–109). ACM. doi:10.1145/1378533.1378552

Knight, T. (1986). An architecture for mostly functional languages.*Proceedings of the 1986 ACM conference on LISP and functional programming* (pp. 105–112). ACM. doi:10.1145/319838.319854

Le, H. Q., Guthrie, G. L., Williams, D. E., Michael, M. M., Frey, B. G., Starke, W. J., Nakaike, T. (2015). Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1), pp. 8:1–8:14.

Leis, V., Kemper, A., & Neumann, T. (2014). Exploiting hardware transactional memory in main-memory databases. *Proceedings of the 2014 IEEE 30th International Conference on Data Engineering (ICDE)* (pp. 580–591). IEEE. doi:doi:10.1109/ICDE.2014.6816683 doi:10.1109/ICDE.2014.6816683

Lev, Y., & Maessen, J.-W. (2008). Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (pp. 197–206). ACM.

Lomet, D. B. (1985). *Process structuring, synchronization, and recovery using atomic actions*. Springer. doi:10.1007/978-3-642-82470-8_21

Marathe, V. J., & Dice, D. (2014). *Lock-clustering compilation for software transactional memory*. Google Patents.

McDonald, A. (2009). *Architectures for Transactional Memory* [Ph.D. Dissertation]. Stanford University, Stanford, USA. Retrieved from http://csl.stanford.edu/~christos/publications/2009.austen_mcdonald.phd_thesis.pdf

Minh, C. C. (2008). *Designing an effective hybrid transactional memory system*. ProQuest.

Moir, M., Moore, K., & Nussbaum, D. (2007). The Adaptive Transactional Memory Test Platform: A tool for experimenting with transactional code for Rock. *Proceedings of the Workshop on Transactional Computing (Transact).* Sun Microsystems.

Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D., & Wood, D. A. (2006, February). LogTM: log-based transactional memory. Proceedings of HPCA '06 (pp. 254-265). doi:doi:10.1109/HPCA.2006.1598134 doi:10.1109/HPCA.2006.1598134

Moore, K. E., & Grossman, D. Log-based transactional memory. *Proc. of the Twelfth International Symposium on High-Performance Computer Architecture* (pp. 11–15).

Moravan, M. J., Bobba, J., Moore, K. E., Yen, L., Hill, M. D., Liblit, B., & Wood, D. A. (2006). Supporting nested transactional memory in LogTM. In *ACM Sigplan Notices* (Vol. 41, pp. 359–370). ACM.

Moss, J. E. B., & Hosking, A. L. (2006). Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, *63*(2), 186–201. doi:10.1016/j.scico.2006.05.010

Navarro, C. A., Hitschfeld-Kahler, N., & Mateu, L. (2014). A survey on parallel computing and its applications in data-parallel problems using GPU architectures.Communications in Computational Physics, 15(2), 285–329.

Negi, A., Titos-Gil, R., Acacio, M. E., Garcia, J. M., & Stenstrom, P. (2012). π-TM: Pessimistic invalidation for scalable lazy hardware transactional memory. *Proceedings of the 2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)* (pp. 1–12). IEEE.

Rajwar, R., & Goodman, J. R. (2001). Speculative lock elision: Enabling highly concurrent multithreaded execution. *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture* (pp. 294–305). IEEE Computer Society.

Rajwar, R., & Goodman, J. R. (2002). Transactional lock-free execution of lock-based programs. *Operating Systems Review*, *36*(5), 5–17. doi:10.1145/635508.605399

Rajwar, R., Herlihy, M., & Lai, K. (2005). Virtualizing transactional memory.*Proceedings of 32nd International Symposium on Computer Architecture ISCA '05* (pp. 494–505). IEEE. doi:10.1109/ISCA.2005.54

Ramadan, H. E., Rossbach, C. J., Porter, D. E., Hofmann, O. S., Bhandari, A., & Witchel, E. (2007). Metatm/txlinux: Transactional memory for an operating system. *ACM SIGARCH Computer Architecture News*, *35*(2), 92–103. doi:10.1145/1273440.1250675

Saha, A., Chatterjee, A., Pal, N., Ghosh, A., & Chaki, N. (2015). A Lightweight Implementation of Obstruction-Free Software Transactional Memory. In *Applied Computation and Security Systems* (pp. 67–84). Springer. doi:10.1007/978-81-322-1988-0_5

Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., & Hertzberg, B. (2006). McRT-STM: a high performance software transactional memory system for a multi-core runtime.*Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (pp. 187–197). ACM. doi:10.1145/1122971.1123001

Scott, M. (2015). Transactional Memory Today. *ACM SIGACT News*, *46*(2), 96–104. doi:10.1145/2789149.2789166

IBM Sequoia. (2015, August 18). Wikipedia, the free encyclopedia. Retrieved from https://en.wikipedia.org/w/index.php?title=IBM_Sequoia&oldid=676692785

Titos-Gil, R., Negi, A., Acacio, M. E., Garcia, J. M., & Stenstrom, P. (2013). Eager beats lazy: Improving store management in eager hardware transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, *24*(11), 2192–2201. doi:10.1109/TPDS.2012.315

Transmeta. (n. d.). *Wikipedia, the free encyclopedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Transmeta&oldid=680746415

Yen, L., Bobba, J., Marty, M. R., Moore, K. E., Volos, H., Hill, M. D., & Wood, D. (2007, February). LogTM-SE: Decoupling hardware transactional memory from caches. *Proceedings of the IEEE 13th International Symposium on In High Performance Computer Architecture* (pp. 261-272). IEEE.

Zilles, C., & Rajwar, R. (2007). Transactional memory and the birthday paradox.*Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures* (pp. 303–304). ACM. doi:10.1145/1248377.1248428

KEY TERMS AND DEFINITIONS

Conflict Detection: In transactional memories, Conflict detection ensures serializability between transactions. It is a process to detect conflicts which must be resolved.

FlexTM: Flexible transactional memories provide a 5× speedup over high-quality software TM.

LogTM: Log-based transactional memories make commits fast by storing old values to a per-thread log in cacheable virtual memory and storing new values in place.

Multithreading: The ability of a CPU or a single core in a multicore processor system to execute multiple processes or threads concurrently.

Shared Memory: A memory that may be simultaneously accessed by multiple programs in order to provide communication among them. Efficient way to pass data between programs.

TCC: Transactional memory Coherence and Consistency is a model in which atomic transactions are always the basic unit of parallel work, communication, memory coherence, and memory reference consistency.

Thread Synchronization: The handshaking or join up of multiple processes at a certain point to commit a certain series of actions.

Versioning: In Transactional Memory, versioning means that code region should appear atomic and all its modifications must be stored and kept isolated from other transactions until the commit time.