An Introduction to Transactional Memory

Jonathan Parri

***Abstract*—Parallel programming has become a complex task as systems enabling such parallelization increase in popularity and scale. Parallel programming can be simplified by presenting an abstraction that allows groups of load and store operations to execute in an atomic fashion. Grouping these load store operations together at an abstracted level forms the basis of transactional memory encompassing concurrency control for memory accesses. Transactional memory is analogous to common database transactions that have been in use for decades for client-server systems. Hardware transactional memory (HTM) and software transactional memory (STM) are the most common types of implementations for transactional memory and have evolved over time to current processor integrated ISA extensions and packaged libraries for many software languages and models.**

***Index Terms*—Transactional memory, atomic operations, software transactional memory, hardware transactional memory.**

I. Introduction

THE concept of a transaction forms the foundation of transactional memory. Transactions first appeared as database unit abstractions with a defined set of attributes. Transactions must appear indivisible and instantaneous to the end-user or observer. The ACID properties developed from [1] were used as a requirement for database transactions, but apply equally to transactions for memory operations.

- **(A) Atomicity**
  Each transaction is atomic, and if part of the transaction fails then the entire transaction fails and the system state is left unchanged.
- **(C) Consistency**
  Any transaction that is performed will take it from one consistent state to another. This is especially imperative when looking at transactional memory where the memory must remain in a consistent state while a transaction has taken place.
- **(I) Isolation** Other transactions or operations cannot access data that has been altered by a transaction currently in progress.
- **(D) Durability** Durability is described as the ability of a system to be able to recover committed transaction updates. In database systems this is especially important with regards to returning to a correct state after a system failure. In the case of transactional memory, it is the weakest requirement. Mainly, we can gather that once a transaction has succeeded, it cannot be lost.

Note that transactions are said to have been *committed* when they have successfully finished. A transaction can either complete successfully and be committed or abort.

Modern database systems allow multiple queries or requests to run in parallel. Furthermore, in the modern database models, programmers need not worry about concurrency. Taking the initiative from the database community, [2] proposed that the database model be extended for computer memory operations, coining the term *transactional memory*. Despite the proposal, no practical implementations were provided and the concept lay dormant for many years until 1993[3], [4].

Despite the parallel programming problem existing for the last few decades, transactional memory was not envisioned as an appropriate solution until the mid-90's and has recently become a highly desirable research topic as multicore systems have become ever prevalent and the dire need for an effective and appropriate parallel exploitation technique has only fueled the research direction. Transactional memory is one possible solution to facilitate parallel programming; however, it requires a shift in the programming paradigm that brings forward new abstractions and ideologies.

The two most common parallel programming models are data parallelism and task parallelism. Data parallelism applies an operation simultaneously to an aggregate of individual items [5]. Data-parallelism is easily exploited in numeric computations. Task parallelism focuses on the distribution of execution processes or threads [6]. Processes or threads require explicit synchronization techniques (semaphores, locks, fork-join). The principal shortcoming of task parallelism is its lack of effective mechanisms for abstraction and composition, which are fundamental to computer science[7].

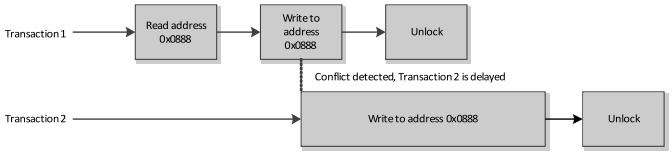
- **Abstraction**
  A simplified view of an entity that captures essential features.
- **Composition**
  Ability to combine entities together to form larger and more complex entities.

Current parallel programming paradigms do not appropriately address abstraction and explicit synchronization cannot be decomposed. Clearly, these fundamental concepts are imperative to furthering the parallel programming camp. Transactional memory provides an abstraction which atomically coordinates concurrent reads and writes. Currently, coordination is the responsibility of the programmer with rudimentary constructs in languages such as Java and C/C++.

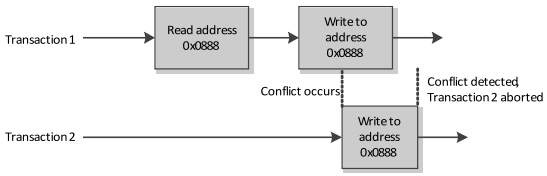
Transactional memory comes in two main implementation formats, software transactional memory (STM) and hardware transactional memory (HTM). Both of these are thoroughly addressed from the implementation and programmer perspectives as this document progresses.

II. Concurrency Control in Transactional Memory

There exist two vital functions found within transactional memory, the detection of a conflict and the resolution of the found conflict. A conflict occurs when two transactions



(a) Pessimistic Concurrency Control.



(b) Optimistic Concurrency Control.

Fig. 1. Concurrency Control Mechanisms.

perform conflicting operations on the same piece of data. With regards to concurrency control, there are two approaches, pessimistic concurrency control and optimistic concurrency control. In pessimistic concurrency control, three events occur at the same time. Here a conflict occurs, it is detected and it is resolved. In this approach the transaction is able to gain exclusive access to a data location prior to the start of the transaction. This prevents other transactions for accessing the already required data of the primary transaction. In optimistic concurrency control, conflict detection and matching resolution can occur after a conflict has occurred. The reasoning behind the *optimistic* idiom is that the system is working under the general assumption that multiple transactions may in fact not attempt to access the same data. Both pessimistic concurrency control and optimistic concurrency control are shown in Fig. 1(a) and Fig. 1(b) respectively.

Note that contention management may be required in the final transactional memory implementation as pessimistic concurrency control may lead to dead lock while optimistic concurrency control to live lock. A variety of contention management possibilities can be implemented[8] but the particular policy depends on the workload and concurrency control used in the transactional memory model. Passive, greedy or round-robin are a few well known contention resolution policies.

Conflict detection is one of the key components of concurrency control. With pessimistic concurrency control, a simple lock is used. When a transaction has finally acquired the lock, conflicts are guaranteed not to occur. For optimistic concurrency control, validation is a common solution. Validation is a simple check that is run on the transaction to verify if any conflicts occurred. Validation can occur during a transaction, at the beginning or at the end commit point[9].

A. Version Control

When dealing with transactional memory, versioning becomes a large consideration. The system must manage concurrent writes from multiple transactions by keeping track of this data. Two versioning approaches are presented in [9], eager version management and lazy version management.

- **Eager Version Management**
  Transactions directly modify data contained in memory, therefore an undo or roll-back log must exist containing the values that were overwritten in the case that the transaction aborts and the system must return to a consistent state. A pessimistic concurrency control model must be used with this form of version management because the transaction requires sole access to the memory locations it wishes to write to.
- **Lazy Version Management**
  In lazy version management, the memory is not updated until a transaction commits. Similarly to the previous version management technique a log is required. This redo-log is used to buffer memory writes. Note that the transaction should verify with the log to ensure data coherence if previous writes have occurred in the transaction. If a transaction is aborted, the log can be flushed.

III. Software Transactional Memory

A. STM Overview

Software transactional memory (STM) is the most common implementation of transactional memory models. STM by nature is more flexible than hardware, easier to change and suffers from fewer bound limitations. For example, an STM is not limited by cache size or address width. The downside of an STM implementation is quite clear, a large overhead.

B. Metadata and Log Files

STM implementations require a way to associate accessed memory locations and concurrency control metadata. Concurrency control metadata is information that allows us to keep track of a transaction. This metadata association takes place in two forms with respect to STMs.

- **Object-Based Association**
  Metadata is generated and managed for and alongside each object that a program allocates. Objects are defined as physical object-oriented objects or blocks of returned memory. Memory allocated for these objects must now be extended to also hold the matching metadata.
- **Word-Based Association**
  Metadata is associated with individual memory locations where a hash function is typically used to map the metadata pieces to the large number of memory locations as it is clearly infeasible to allocate and maintain metadata for every word in memory.

Each approach has its merits but we must consider: the effect of the metadata on the volume of memory used, the effect of accessing the metadata on the performance of the program, the speed of mapping a location to metadata and the likelihood of false conflicts between concurrent transactions[7]. It is interesting to note that sometimes it may be advantageous to use the object-based association as the metadata we are looking for may be on the same cache line. Also, one piece of metadata may cover many fields within a single object and conflicts are inherently explicit as they are managed by the programmer and not the system's runtime. Conflicts can occur with object-based associations when different fields in an object require concurrent access.

If an STM is using eager version management then an undo-log is required. The undo-log stores the values to be restored to memory if a transaction aborts and a rollback is required. On the flip side, if lazy version management is used then redo-logs are required where values are stored until they can be written to memory once the transaction commits. The design of a redo-log is a performance critical issue. A transactional read needs to see the results of earlier transactional writes to the same locations. Using a basic iterative approach, a transaction that writes N times requires up to $O(N)$ for each read in the transaction. Three methods have been proposed to alleviate this iterative searching:

1) **Auxiliary Look-Up Table**
   Provides mapping from an accessed address to a previous redo-log entry[10].
2) **Bloom Filter while Maintaining Write-Set** The log is searched only when reading from a location that might be found in the write-set using a summary log and a Bloom filter[11].
3) **Links to Redo-Log Entries** Links can be provided in the metadata to redo-log entries that are currently being written to in the transaction.

C. Lock-Based STM Systems

1) Local Version Numbers: Lock-Based STM Systems that use local version numbers combine the concepts of pessimistic concurrency control for writes and optimistic concurrency control for reads. Locks are acquired dynamically by the STM and are unknown to the programmer. Version numbers adhere to individual objects and are checked during validation. As the version numbers are local, they are incremented independently for each metadata block once an update has been committed.

Both eager and lazy version management can be used in this approach. Furthermore, the point when locks are acquired is also up for contention. If a lock is acquired when a transaction first accesses a memory location then either version management technique can be used. If a lock is only acquired when a transaction commits, then only lazy version management can be used. This is because the STM cannot update the actual memory until a lock has been acquired. Local version approaches are further analyzed in [12].

2) Global Clock: A global clock implementation is a process-wide incremental counter. The use of a global clock provides opacity of the transaction without requiring incremental validation[13]. Versioned locks are acquired dynamically once a commit has taken place. These versioned locks hold either a time-stamp or identify a transaction whom is the owner of the associated data. In [13], lazy version management is done so a redo-log is required. A simple 64-bit shared counter is incremented everytime a transaction commits.

Transactions begin by reading the current global clock value. The value returned is used as a read version time-stamp and is also used a write version indicating a transaction's order with respect to all other transactions. The complete act showing atomicity is shown in Fig. III-C2.

3) Global Metadata: In lock-based STM systems with global metadata there are no individual locks or version numbers associated with objects only a shared global structure containing metadata. Such an implementation avoids the space overhead and cache pressure of many scattered metadata fragments. The two main considerations of such an implementation are: the design should access the metadata sparingly to reduce memory contention as metadata is shared and the detection of conflicts in terms of locations accessed by a transaction.

Bloom filters can again be used for conflict detection as they provide a good estimate of a set's contents[14]. Both insertion and query operations of a Bloom filter occur in $O(1)$. We can query all indices in the filter and see if they are all set. Bloom filter conflict detection in a lazy-versioning STM was first introduced in [15].

The second conflict detection technique is value-based validation. A transaction can verify if it has experienced a conflict by comparing a log of actual values it has read from memory. This technique is implemented in [16].

D. Locking Problems with Transactional Memory and Non-Transactional Operations

As some back-end STM libraries do in fact utilize locks, unbeknown to the programmer, problems can occur. The reader is reminded that an STM model with pessimistic concurrency control is using a hidden lock within the supporting logic. The purpose of transactional memory is to not concern the programmer with such synchronization primitives but have them taken care of automatically in the STM framework, as such, common locking problems can arise. A taxonomy of lock-based synchronization issues within the scope of transactional memory and non-transactional operations is shown below[17]:

- **Non-Repeatable Read**
  If A transaction reads the same variable multiple times and a non-transactional write occurs in between these writes, a non-repeatable read can occur. Buffers can be used to store first accessed data that is read during a transaction if it is possible that a non-transactional entity may alter the memory space.
- **Intermediate Lost Update**
  A non-transaction write can be lost and not seen by the transaction if a non-transactional write comes between a read-modify-write operation series executed by a transaction. A Bloom filter is used to test whether an element is in a set based on a probabilistic data structure. Note that false positives can occur, but false negatives cannot.
- **Intermediate Dirty Read**
  If an STM is using eager version management where a non-transactional read sees an intermediate value written by a transaction instead of the committed data.

E. Non-Blocking STMs

So far, all the STMs discussed have used a back-end opaque locking mechanism in the transactional model. It is possible for STMs to provide non-blocking progress guarantees. Such progress guarantees may include obstruction or lock freedom. The challenge with non-blocking transactional memory models is in ensuring that memory accesses appear to be occurring atomically despite the transaction performing many individual

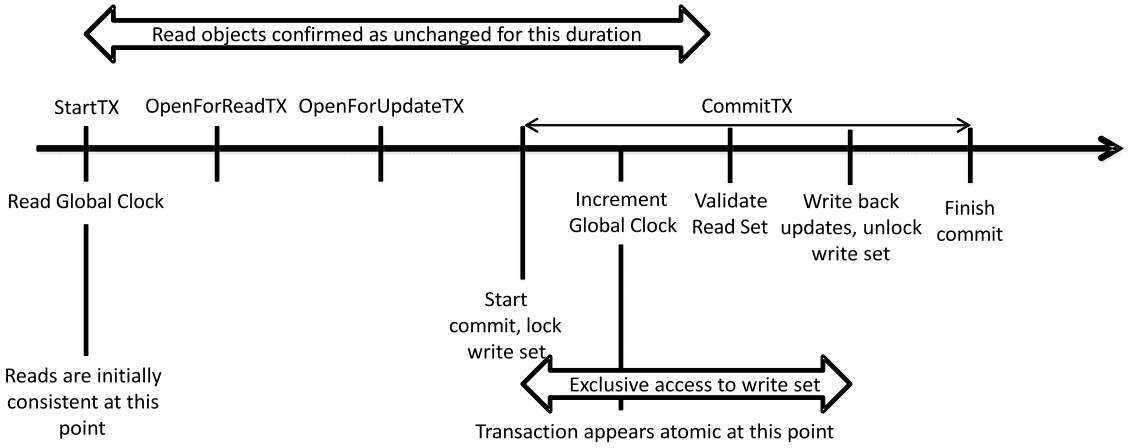


Fig. 2. Atomicity with Global Clock. Adapted from [7].

atomic operations. Mutual exclusion is used to accomplish this with lock-based systems as all locations needed are locked.

The first non-blocking STM was coined as a dynamic software transactional memory (DSTM)[18]. The STM was dynamic in the sense that the programmer need not specify memory locations that a transaction wished to access in advance. Such a concept was implemented as a Java library offering explicit transactions as opposed to traditional atomic blocks. Two forms of indirection were used on a per-object basis.

Newer techniques have looked to move away from indirection; however, [18] proved very influential with future works. The three concepts that were introduced and are are still important in more recent work include[7]:

1) Dynamic association of objects with a transaction descriptor allowing a single update to a descriptor's status field.
2) Definition of an object's logical contents in terms of metadata and object snapshots.
3) Use of immutable shared data for locators where locks are not needed.

A taxonomy of potential design decisions with regards to non-blocking STMs was created in [19] and summarized in [7]. These taxonomy questions are presented below:

- When are objects acquired for writing?
- Are readers visible to writers?
- How are objects acquired by transactions?
- How do objects appear when they are inactive?
- Which non-blocking progress property is provided? emphie. lock-freedom or obstruction-freedom

Similarly to lock-based designs, it is possible to have a non-blocking STM avoid the use of indirection for an object's representation in memory. The first of such a system was [20] where a word-based system was used and a separate table was present which was associated to the program heap using a hash function. Each transaction's metadata held tuples for each location that the transaction accessed.

IV. Hardware Transactional Memory

A. HTM Overview

There are plenty of hardware transactional memory(HTM) models that have been proposed. In actuality, real implementations are finally slowly being considered as viable options. HTM systems offer several advantages over the previously examined STM implementations: lower overhead, less invasive to code-base, possibly decreased power usage, provide strong encapsulation without changing the way non-transactional memory accesses occur and the effectiveness of HTMs for languages without dynamic compilation and garbage collection. Despite implementation differences, many of the concepts presented for STMs carry over to the hardware realm. HTMs must again identify locations for transaction accesses, manage read and write-sets of transactions, detect and resolve data conflicts, manage architectural register state and be able to commit or abort transactions.

To identify transactional memory accesses, instruction set extensions are used, that is, special transactional operations added to a processor's instruction-set architecture (ISA).

B. HTM Categories

There exists two categories of HTM models and implementations.

1) **Explicitly Transactional**
   Provide software with new memory instructions to show which memory accesses are to be made transactionally.
2) **Implicitly Transactional**
   Software need only specify the boundary of a transaction. All memory accesses within the boundary are transactional.

C. Tracking Reads and Writes

Similar to STMs, an HTM must keep track of a transaction's read-set and write-set. HTMs utilize available caches and buffers to complete this tracking obligation thus lowering needed computation overhead. The data-cache can be used to track a transaction's read-set as all memory reads naturally

read first from this cache. One option is to duplicate the data-cache to keep track of reads. This concept was introduced in [21] as a parallel transactional cache but added significant overhead and complexity to the system. It is possible to extend the existing data-cache for tracking read-sets. Typically a *read bit* is added to each cache line. Once a transactional read occurs, the *read bit* is set showing that a particular cache line is in a transaction's read set. Sun's Rock processor[22] uses this data-cache implementation to track read-sets. Note that by using this data-cache method, the granularity of conflict detection is the size of a cache line.

The data-cache can also be extended to track the write-set involved in a transaction. A speculatively written state[23] is added for all addresses involved in the transaction. We must ensure that exclusive cache line copies are not lost in systems where a write-back cache policy is used. The dirty cache line must be written back to memory before it is overwritten in the cache.

Other proposals have introduced the addition of dedicated buffers to track read-sets and write-sets[24], [4]. It has been shown in [25] that the use of the data-cache to track both read and write-sets is quite effective, supporting well over 30000 instructions within transactions operating on hundreds of cache lines.

D. Conflicts in the HTM

Data conflicts need to be detected when the read and write-sets of concurrently executing transactions overlap, attempting to access the same memory locations. Conflicts within buffers or caches are detected using standard cache coherence mechanisms such as M(O)ESI[26].

E. Conflict Resolution

Most HTM systems and models perform eager conflict detection where the transaction on a processor that receives a conflicting request is aborted. At this point, control is handed over to a software handler. The handler then can reissue the transaction or perform contention management.

F. Register States

Only the state of the memory and cache has been discussed thus far. It is imperative to consider the internal architectural registers within the system as well. As Fig. 3 shows, the register and memory state form a system's precise state. If a transaction is to be aborted, the registerfile and control registers must be rolled back along with the memory state.

There are hardware and software solutions to this roll-back problem and are listed next.

1) Rely on software for recovery of state from log files.
2) Use shadow registers to restore state.
3) Exploit architectures with speculative execution.
4) Hybrid approach where some registers are restored by software and others by hardware (*ie.* Sun's Rock Processor[22].

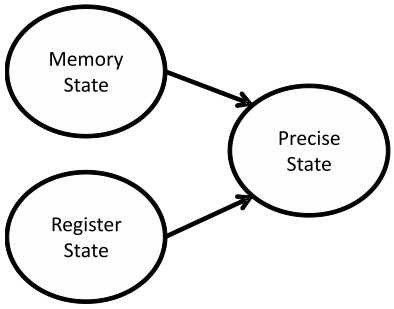


Fig. 3. Precise State and its Components.

G. A Verifiable HTM Proposal and an Implementation

1) Sun's Rock Processor[22]: The Rock Architecture presented by Sun in 2008 and produced in 2009 is an implicitly transactional architecture. The extensions to the ISA indicate the beginning and end of a transaction. The two new instructions are as follows:

`chkpt<fail-address>`

- Instruction used to specify an address to go to following an abort and where the transaction begins. The specified address should contain a software handler to deal with an aborted transaction.

`commit`

- Instruction used to indicate where a transaction ends.

All operations between the two above instructions are implicitly transactional. In this architecture, the data cache is used to track the read-set while the store buffer is used for the write-set. The L2 cache is required to store all addresses to be used inside a transaction. When a commit is executed, the L2 cache locks all cache lines corresponding to the tracked store addresses. The store operations drain the store buffer, update the cache, then release the lock preventing conflicting access during the commit sequence. A special register has been added which stores a flag specifying why a transaction was aborted. This data can be used by the software handler. The architectural register states can be updated in 1 clock cycle using a custom flash-copy mechanism.

2) AMD's Advanced Synchronization Facility[24]: AMD has recently proposed the addition of transactional memory to the x86-64 architecture. They have proposed an explicitly transactional model, Advanced Synchronization Facility, that is currently being evaluated in PTLSim[27]. The new instructions available in the ISA are detailed next.

`SPECULATE`

- Instruction used to begin a transaction.

`COMMIT`

- Instruction used to indicate where a transaction ends.

`LOCK`

- Instruction for memory access that needs to be performed transactionally.

Control is returned to the speculative region if a transaction aborts. Appropriate processor flags are then set to indicate why an abort occurred. In this architecture, dedicated registers used to perform multi-word compare-and-swap like instructions have been added. As for the architectural register state, the architecture only supports recovering the stack pointer with hardware support, the rest of the registers rely on software.

V. Partial Re-execution of Transactions

If a transaction aborts there can be a large penalty due to the time wasted by having the transaction start from scratch again. It has been suggested that intermediate checkpoints[28] be placed throughout a transaction's execution holding various snapshot states as the transaction progresses. Another possible solution involves the nesting of transactions within transactions to create these progression snapshots. This method is known as abstract nested transactions[29].

VI. Conclusion

Transactional memory as a new programming abstraction has recently become a hot research topic. This abstraction allows parallelization opportunities presented in new systems to be better exploited by shifting the existing programming paradigm.

Unfortunately, after the acquisition of Sun by Oracle, it looks as though the only readily available HTM, Rock Processor, has been removed from future development cycles and support. Hopefully, AMD can become the next leader in HTM as it further develops its Advanced Synchronization Facility. STM libraries are abundant for a variety of languages (*ie.* Java, C++) but until a standardized model is created that everyone can agree on, software transactional memory will most likely stay as an academic research initiative. The hybrid approach of taking the best parts of STM and HTM is a topic of interest that is currently in the limelight and may lead to better acceptance in the future. STM and HTM complement each other more then they compete.

Academics still believe that transactional memory will be the future programming mechanism of multicore and multiprocessing environments. It is up to industry to make the next step.

Appendix A
Questions about transactional memory

1) *Why must a pessimistic concurrency control model be used with eager version management?*
   Transactions with eager version management require sole access to memory locations that they wish to write to.
2) *What are some complexity trade-offs with regards to current HTM proposals?*
   The introduction of an HTM requires massive changes to existing tool-chains and system libraries. The use of an HTM may also affect security, performance and reliability requirements of a system and should be thoroughly analyzed.
3) *The idea of transactional memory is that primitive synchronization mechanisms are not used by the user (ie. locks). Is it possible that locks are actually used?*
   Many STM/HTM solutions actually implement a lock, blocking, in the back-end. These locks are not seen by the user, opacity, but are dynamically acquired by the transaction mechanism. There exist also non-blocking solutions for both STMs and HTMs that do not use an atomic lock.
4) *HTM proposals discussed in this paper have been bounded, meaning that the transactions are assumed to fit within hardware buffers. What challenges are present if an STM is used as a fall-back mechanism for an HTM that has exceeded its resource allowance?*
   Synchronization and decoupling between the two mechanisms becomes difficult. Solutions may include two memory structures for tracking metadata and the state of both the HTM and STM.
5) *List one way that it is possible to expose HTM functionality to an STM?*
   Perform conflict-detection using the extended data-cache and continue using metadata and logs for other transactional attributes.
6) How are conflicts detected in an HTM?
   Simple cache coherence protocols (MESI) are used.

Appendix B
Load-Link/Store Conditional Analysis

Load-link/store-conditional(LL/SC) implements a lock-free atomic read-modify-write operation. The LL returns the current value of a memory location while the SC stores a new value only if no updates have occurred to the destination location since the LL. LL/SC is considered one of the most basic transactional memory operations. Note that typically LL/SC only operates on data that is the size of a native machine word and is therefore inherently bound.

Appendix C
Transactional Memory Simulators

- (2010) PTLSim augmentation with AMD64 architecture extension for transactions.
  http://www.amd64.org/research/multi-and-manycore-systems.html

- (2009) Deuce STM provides support for transactions within the JVM.
  http://www.deucestm.org

- (2010) Dresden TM Compiler supporting transactions in C/C++.
  http://tm.inf.tu-dresden.de

- (2008) IBM XL C/C++ for Transactional Memory Compiler introduces pragma based atomic specification

in C/C++.
http://amino-cbbs.sourceforge.net

- (2009) Intel C++ STM Compiler with C++ extensions. http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition

- (2008) Java library implementing STM with partial re-execution of failed transactions. http://wen.ist.utl.pt/~joao.cachopo/jvstm

- (2007) MetaTM is a hardware transactional memory simulator for the Virtutech Simics platform. http://www.metatm.net

REFERENCES

[1] (1998, November) Gray to be Honored With A. M. Turing Award This Spring. Microsoft PressPass. [Online]. Available: http://www.microsoft.com/presspass/features/1998/11-23gray.mspx

[2] D. B. Lomet, "Process structuring, synchronization, and recovery using atomic actions," in *ACM Conference on Language Design for Reliable Software*, March 1977, pp. 128–137.

[3] M. Herlihy, J. Eliot, and B. Moss, "Isca 93: Proc. 20th annual international symposium on computer architecture," in *Transactional Memory: Architectural Support for Lockfree Data Structures*, May 1993, pp. 289–300.

[4] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek, "Multiple Reservations and the Oklahoma Update," *IEEE Parallel & Distributed Technology*, pp. 58–71, 1993 1993.

[5] D. Hillis and G. Steele, "Data parallel algorithms." *Communications of the ACM*, pp. 1170–1183, 1986.

[6] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*. Addison Wesley, 2005.

[7] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory: Synthesis Lectures on Computer Architecture*, 2nd ed. Morgan Claypool, 2010.

[8] R. Guerraoui, M. Herlihy, and B. Pochon, "Toward a theory of transactional contention managers," in *PODC 05: Proc. 24th Annual ACM SIGACTSIGOPS Symposium on Principles of Distributed Computing*, July 2005, pp. 25–264.

[9] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood, "LogTM:Log-based transactional memory," in *12th International Symposium on High-Performance Computer Architecture*, 2006, pp. 254–265.

[10] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi, "Optimizing memory transactions," in *PLDI 06: Proc. 2006 ACMSIGPLAN Conference on Programming Language Design and Implementation*, June 2006, pp. 14–25.

[11] B. O'Sullivan, J. Goerzen, and D. B. Stewart, *Real World Haskell*. O'Reilly Media, Inc., 2008, ch. Advanced Library Design: Building a Bloom Filter.

[12] D. Dice and N. Shavit, "What really makes transactions faster?" in *TRANSACT 06: 1st Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006, p. June.

[13] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," in *DISC 06: Proc. 20th International Symposium on Distributed Computing*, September 2006, pp. 194–208.

[14] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, pp. 422–426, 1970.

[15] M. F. Spear, M. M.Michael, and C. von Praun, "Ringstm: scalable transactions with a single atomic instruction," in *SPAA 08: Proc. 20th Annual Symposium on Parallelism in Algorithms and Architectures*, June 2008, pp. 275–284.

[16] M. Olszewski, J. Cutler, and G. Steffan, "Pact 07: Proc. 16th international conference on parallel architecture and compilation techniques," in *JudoSTM: a dynamic binaryrewriting approach to software transactional memory*, September 2007, pp. 365–375.

[17] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. Hudson, K. F. Moore, and B. Saha, "Enforcing isolation and ordering in stm," in *2007 ACMSIGPLAN Conference on Programming Language Design*, June 2007, pp. 78–88.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III, "Podc 03: Proc. 22nd acm symposium on principles of distributed computing," in *Software transactional memory for dynamic-sized data structures*, July 2003, pp. 92–101.

[19] M. F. Spear, V. J. Marathe, W. N. S. III, and M. L. Scott, "Conflict detection and validation strategies for software transactional memory," in *DISC 06: Proc. 20th International Symposium on Distributed Computing*, September 2006.

[20] T. Harris and K. Fraser, "Language support for lightweight transactions," in *OOPSLA 03: Proc. Object-Oriented Programming, Systems, Languages, and Applications*, October 2003, pp. 388–402.

[21] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *ISCA 93: Proc. 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 289–300.

[22] (2008) Rock: A SPARC CMT Processor. [Online]. Available: http://www.opensparc.net/pubs/preszo/08/RockHotChips.pdf

[23] C. Blundell, J. Devietti, C. Lewis, and M. M. K. Martin, "Making the fast case common and the uncommon case simple in unbounded transactional memory," *SIGARCH Computer Architecture News*, pp. 24–24, 2007.

[24] C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere, "Evaluation of amds advanced synchronization facility within a complete transactional memory stack," in *EuroSys 10: Proc. 5th ACMEuropean Conference on Computer Systems*, April 2010.

[25] R. Rajwar and J. R. Goodman, "Speculative lock elision: enabling highly concurrent multithreaded execution," in *MICRO 01: Proc. 34th International Symposium on Microarchitecture*, December 2001, pp. 294–305.

[26] P. Sweazey and A. J. Smith, "A class of compatible cache consistency protocols and their support by the IEEE Futurebus," in *ISCA 86: Proc. 13th annual international symposium on Computer architecture*, 1986, pp. 414–423.

[27] (2010) PTLSim augmentation with AMD64 architecture extension for transactions. [Online]. Available: http://www.amd64.org/research/multi-and-manycore-systems.html

[28] E. Koskinen and M. Herlihy, "Checkpoints and continuations instead of nested transactions," in *SPAA 08: Proc. 20th Annual Symposium on Parallelism in Algorithms and*, June 2008, pp. 160–168.

[29] T. Harris and S. Stipic, "Abstract nested transactions," in *InTRANSACT 07: 2nd Workshop on Transactional Computing*, August 2007.