INFORMATION TECHNOLOGY JOURNAL

Information Technology Journal 9 (1): 192-200, 2010
ISSN 1812-5638
© 2010 Asian Network for Scientific Information

A Review of Transactional Memory in Multicore Processors

X. Wang, Z. Ji, C. Fu and M. Hu
Harbin Institute of Technology, P.O. Box 1209, No. 13 Fa Yuan Street, 150001 China

Abstract: To develop composable parallel programs easily and get high performance, many transactional memory systems have been proposed to solve the synchronization problem of multicore processors. Transactional memory can be implemented in hardware, software, or a hybrid of the two. There are many hot topics in current transactional memory systems. In this study, we give a review of the current transactional memory systems for multicore processors according to the following aspects: version management, conflict detection and contention management. Then we separately present exclusive research area for hardware and software transactional memory. Finally, we separately discuss research challenges for software and hardware transactional memory.

Key words: Multicore processor, transactional memory, parallel programming, synchronization

INTRODUCTION

Multicore processors are now prevailing in server, desktop and even embedded systems. However, it is very difficult to develop parallel programs for processors with increasing number of cores. Because application developers have to face with burdens such as synchronization tradeoffs, deadlock avoidance and races. In this environment, Transactional Memory (TM) has been proposed as a new parallel programming model that is easy and efficient for writing parallel programs (Herlihy and Moss, 1993).

Transactional memory is used to replace critical sections protected by locks in multi-threaded parallel programs by transactions (atomic blocks). Compared to critical sections, transactions have several advantages. First, programmers are liberated from reasoning about the correctness and performance of their locking scheme. Second, shared data structures are guaranteed to be kept in consistency even in the event of a failure. Third, transactions can be composed naturally, that make it much easier for developing composable parallel software.

The concept of transaction is firstly used in the database research area. Like database transaction, TM has Atomicity, Consistency and Isolation (ACI) properties: Atomicity to ensure all of the transaction's instructions either commit successfully or abort, Consistency to ensure all transactions view a single completion order across the whole system and Isolation to ensure that each transaction's operations are not visible to other transactions, no matter how many transactions are concurrently executing.

If there are no conflicts, TM systems can execute multiple transactions in parallel. If two transactions access the same memory item and at least one of them writes, they are conflicted. In this case, one of them is aborted and restarts. As a transaction starts, it checkpoints registers to save old values, which can be restored in case of abort. A transaction cannot write to shared memory directly; instead its results are stored in an undo-log or a write-buffer maintained by system. In order to detect read-write or write-write conflicts, memory references are tracked. If a transaction completes without conflicts, its results are committed to shared memory. If a conflict is detected between two transactions, one of them rolls back by restoring the register checkpoint.

Transactional memory can be implemented in hardware (Wang *et al*., 2009), software (Fu *et al*., 2009), or a hybrid of the two. Software Transactional Memory (STM) systems (Harris and Fraser, 2003; Herlihy *et al*., 2003a; Larus and Rajwar, 2006; Saha *et al*., 2006; Shavit and Touitou, 1995) is the easiest to implement, requiring no changes to existing hardware. But for most STMs, poor performance and weak atomicity are two serious disadvantages. According to two research results (Harris *et al*., 2006; Tabatabai *et al*., 2006) , even though the code can be optimized by compilers, STM can still slow down each thread by 40% or more. More severely, most high-performance STM systems support only weak atomicity (Blundell *et al*., 2005), which guarantees transactional semantics only among transactions. Weak atomicity may produce incorrect or unpredictable results even for simple parallel programs that would work correctly with lock-based synchronization (Dice and Shavit, 2007; Larus and Rajwar, 2006; Shpeisman *et al*., 2007). As a result, supporting only weak

Corresponding Author: Xiaoqun Wang, Harbin Institute of Technology, P.O. Box 1209, No. 13 Fa Yuan Street, 150001 China
Tel: +86-451-86413354

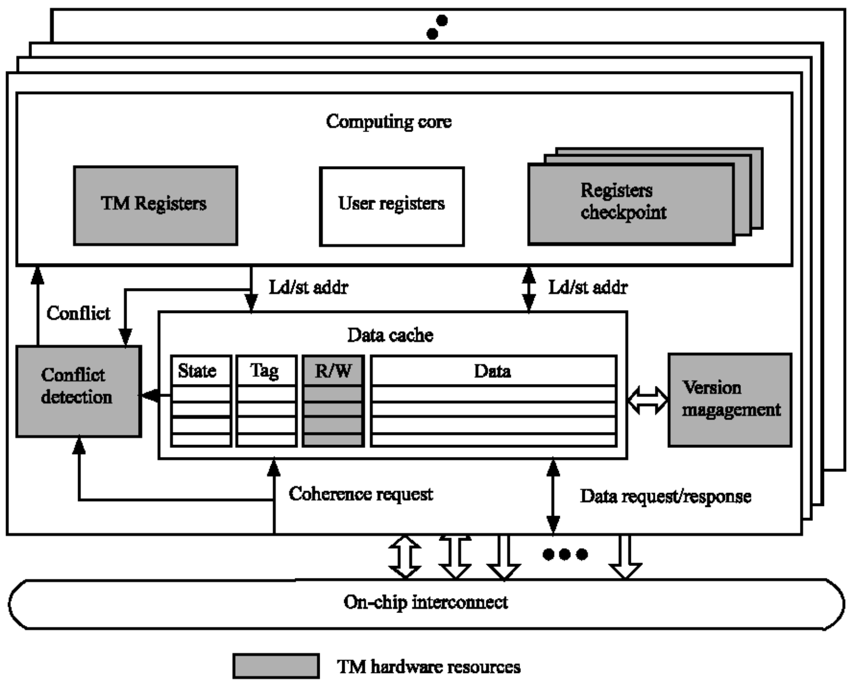

Fig. 1: A framework of hardware transactional memory system

atomicity will likely make writing and debugging more challenging, undermining the primary benefit of transactional memory.

Compared to STM, Hardware Transactional Memory (HTM) naturally has the advantages of high performance and strong atomicity. Typically, HTM systems use hardware caches to track the data read or written by each transaction and leverage the cache coherence protocol to detect conflicts between concurrent transactions (Hammond *et al*., 2004; Moore *et al*., 2006). By using hardware, HTM systems eliminate the overhead of acquiring and releasing fine-grained locks, so they have higher performance not only than STMs, but also than lock-based synchronization for most applications. In addition, by leveraging the cache coherence mechanism, they naturally check any memory access of any active transactions. Thus, they provide strong atomicity with little or no additional overhead. While there are several HTM schemes, most HTMs are quite similar in their structure. Figure 1 shows a framework of HTM system. Register values are stored in extra register files in the beginning of transactions in case of abort. To track transactional memory accesses, a pair of read and write bits per cache line are added. Cache is used as buffer for transactional data and performs version management. Cache coherence protocol mechanism is integrated with conflict detection. Memory requests from the other cores are snooped and checked against the read/write bits for conflict detection. Version management is invoked for transaction commit, abort and conflict.

VERSION MANAGEMENT

Version Management is the mechanism to deal with the different versions of a logical data: the new updated versions from different transactions and the old version for rollback to the original data in case a transaction aborts. Generally, there are two kinds of version management: Lazy Version Management (LVM) and Eager Version Management (EVM).

In STM systems with LVM, such as TL, OSTM, WSTM, DSTM, ASTM, RSTM, Haskell STM (Dice and Shavit, 2006; Fraser, 2004; Harris and Fraser, 2003; Harris *et al*., 2005; Herlihy *et al*., 2003b; Marathe *et al*., 2006; Perfumo *et al*., 2008), old version is remained in its original place and new versions are stored in a per-transaction buffer. When a transaction commits, a new version replaces the old version and the new version's address in store buffer is released. When a transaction aborts, the new version in store buffer is discarded directly. Therefore, LVM is more efficient for transactions aborting. In addition, multiple transactions can concurrently access a shared object, with each of them keeping a private version of the object in store buffer and no one committing at the time. Hence, LVM allows concurrent transactional read and write for the same logical data.

In STM systems with EVM, such as Bartok STM, Autolocker, McRT-STM (Harris *et al*., 2006; McCloskey *et al*., 2006; Saha *et al*., 2006), new version replace old version directly, while backup the old version in a checkpoint (Li and Yang, 2000). Compared with LVM, EVM reduces the copy cost in LVM, because the new version data is stored in the old version's address, thus only a new version can be stored. However, it prevents other transactions to read a modified uncommitted object, thus limits the possible concurrency. With EVM, a transaction's committing is simple: just discarding the old version in its checkpoint. While a transaction aborts, the old version in its checkpoint is restored to its original place and the new version is discarded. Therefore, EVM is more efficient for transactions committing, especially when transactions commit more frequently.

HTM systems also have LVM and EVM. Compared to STM systems, Version management in HTM systems often depend on augmenting processor caches (Wang *et al.*, 2005, 2006a, b). Lazy version management puts old values in memory for making fast aborts (Ananian *et al.*, 2005; Ceze *et al.*, 2006; Hammond *et al.*, 2004; Rajwar *et al.*, 2005). Conversely, eager version management stores newly values in target address for making fast commits and slow aborts (Moore *et al.*, 2006; Moravan *et al.*, 2006; Yen *et al.*, 2007).

CONFLICT DETECTION

A conflict happens when two transactions access one logical data and at least one modifies the data. When a conflict is detected, the STM system or the related transactions will take some method to resolve it, normally by aborting or deferring one related transaction.

Generally, there are three type of conflict detection in STM systems: Eager Conflict Detection (ECD), Lazy Conflict Detection (LCD) and Hybrid Conflict Detection (HCD).

ECD which is used in HICKS, AUTOLOCKER and Shavit STM (Hicks *et al.*, 2006; McCloskey *et al.*, 2006; Shavit and Touitou, 1995) detects conflicts when a transaction wants to access memory while LCD detects conflicts when a transaction is about to commit updates. ECD always works with EVM, since it is necessary to make sure that only one transaction can write a new version to a logical data. Thus, the system must detect conflicts first. Similarly, LCD which is used in TL, OSTM, SXM, WSTM, DSTM, ASTM, RSTM and Haskell STM (Dice and Shavit, 2006; Fraser, 2004; Guerraoui *et al.*, 2005a,b; Harris and Fraser, 2003; Herlihy *et al.*, 2003a, b; Marathe *et al.*, 2005; Marathe *et al.*, 2006; Perfumo *et al.*, 2008) in commonly works with LVM, since all updates are private and invisible to others, which do not need conflict detection before committing. Some HTM systems provide the combination of LVM and ECD, which is rarely used in STM systems. On the other hand, EVM cannot work with LCD together, because only one new version can be stored for a logical data, therefore conflict detection must process as soon as possible to ensure only one transaction can write the new version to the location of the logical data.

Some STMs use HCD, which combines ECD and LCD. For example, McRT-STM (Saha *et al.*, 2006) and Bartok STM (Harris *et al.*, 2006) which manage transactional version in EVM mechanism use ECD before a transaction modify an logical data, but allow multiple transactions to read a shared data concurrently and to delay detecting read-write conflicts until committing.

Compared STMs, HTMs depend on cache coherence protocols to detect conflict (Moore *et al.*, 2006; Yen *et al.*, 2007). In general, cache coherence protocols have no flexibility like STM implementation. And they are difficult to verify (Plakal *et al.*, 1998; Sorin *et al.*, 2002). But they have higher performance than STM implementation. Another difference between HTM and STM conflict detection is granularity for detection. HTM systems use cache lines as granularity (Hammond *et al.*, 2004). STM systems have three granularity choices in reality implementation: word (Shavit and Touitou, 1995) (or block), object and hybrid. STM object granularity have more flexibility for different languages, especially for object-based languages (Guerraoui *et al.*, 2005b). Hybrid granularity in STM can change according to practice requirements.

CONTENTION MANAGEMENT

In TM, Contention Management (CM) refers to the mechanism to determine which transaction involved in a conflict should abort or stall and when the aborted or stalled transaction should be restarted. When a conflict happens, the involved transactions are divided into two sides: the attacker and the defender. The attacker is the transaction requesting access to a shared memory while the defender is the transaction receiving the attacker's request. In eager conflict detection systems, the attacker is the transaction requesting a load for a shared memory. In lazy conflict detection systems, the attacker is the completing transaction attempting to validate that no other transactions conflict with it.

When conflicts happen, the decision of which one of the attacker and the defender to win is generally coupled with the conflict detection and version management policies. For examples, the attacker always wins in LL systems while either attacker or defender wins in EE systems. After the winner has been selected, there are two methods to handle the other transactions: stalling or aborting.

As for stalling, it does not abort the non-winner transactions but stalls them until the winner has finished its transaction. Therefore, it not only saves the work done by non-winner transactions but also saves the expensive cost for unrolling the undo log in eager version management system. Although, stalling can save more time than aborting, it easily leads to deadlock: A reads X, B reads Y, A tries to write Y, stalls on B, while B tries to write X and stalls on A. Deadlock can be solved by these traditional deadlock avoidance methods such as Greedy and LogTM (Guerraoui *et al.*, 2005a, b; Moore *et al.*, 2006). In addition, the stalling method requires global knowledge of commit and abort events. It is very difficult to implement such a system with that global knowledge because it requires significant modifications to existing cache coherence protocols and/or virtual memory systems. For this reason, some practical systems just stall non-winner transactions by waiting for a fixed time.

Compared with stalling, the aborting method is more easily to implement and is free from deadlock, thus it is widely used especially in EE systems. In these systems, two questions need to answer when aborting a transaction. The first is what kind of a transaction will be aborted. The second is when restart a transaction after aborting.

For choosing a transaction to abort, there are a number of measures, such as ages of the transactions, size of read-and write-sets, the number of previous aborts, etc. The age of a transaction is a simple method that aborts the newer transaction in a conflict (Rajwar and Goodman, 2002). BulkTM uses this scheme (Ceze *et al.*, 2006). In MetaTM the number of read-written words is used as the priority. The oldest transaction is voted in all aborted transactions after a fixed number of aborts. Karma (Schere III and Scott, 2004) is similar, using the size of the write-set as priority. Of course, a policy named Randomized (Scherer III and Scott, 2005) does not need to consider that which measure is the best indicator for choosing the aborting transaction in so many measures. The only thing is choose to randomly abort the attacker or defender.

Restarting a conflicting transaction could bring another conflict immediately. It indicates that interconnect bandwidth is refilled by cache misses. To avoid this phenomenon, linear, exponential and randomized backoff policies are used in the practice systems. The linear policy is commonly used, but can lead to livelock. The randomized backoff combined with randomized aborting in some practice systems. The Polite (Herlihy *et al.*, 2003a, b) policy employs exponential backoff before restarting conflicts. For preventing starvation, it guarantees transaction success after some n retries.

EXCLUSIVE TOPIC FOR HTM

Virtualization: Many sizes such as transaction size, transaction read-sets and write-sets size in present TM workloads are becoming larger and larger. It makes hardware resources on chip limited. To allow TM to be integrated with other transactional programming models, such as databases, file systems, or message queues, the future workloads are expected to support I/O and blocking system calls within atomic blocks of code. It is very important to deal with some applications with transactions that beyond the limits of hardware resources. Transactions must not be limited to the physical resources of any specific hardware implementation.

Early HTM systems, such as HMTM (Herlihy and Moss, 1993), TCC, UTM, LogTM and Bulk, maintain TM state in structures tightly coupled to the processor caches. These systems execute programs with small transactions even more efficiently than that with lock-based synchronization. But they fail for or lower performance for larger transactions exceeding cache size. In these systems, HMTM can only support transactional memory limited in cache or buffer size. TCC enters a nonspeculative mode if an overflow occurs. UTM and LTM are the first unbounded HTM system proposed, use a local uncached memory region as extra storage for cache overflows. This mechanism requires non-trivial hardware extensions, including a virtual address pointer added to each block in memory (also requiring address translation logically at memory). Bulk uses signatures to encode read-sets and write-sets, making transactions in it can access any number of cache blocks without serializing transactions. LogTM modifies the coherence protocol to allow transactional state to escape the cache.

More recently, HTM systems focused on addressing the problem of virtualizing transactional states across time. To solve the problem, VTM uses a new data structure (the XADT) placed in virtual memory. When space or time virtualization happens, the XADT is used to hold the cache blocks accessed by transactions. For further optimization, VTM adds dedicated hardware to accelerate processing of the XADT. XTM (Chung *et al.*, 2006) and PTM (Chuang *et al.*, 2006) utilize pages from the virtual memory systems to handle transaction overflows, requiring significant modifications to already-complex virtual memory systems. Small hardware signatures (e.g., 2 Kbit) can be moved around on context switch and paging events, thus are easy to be virtualized. However, signatures may lead to false conflicts (Zilles and Rajwar, 2007), which degrade performance by unnecessarily serializing non-conflicting large transactions increase the probability of false conflicts, leading to a further worsening of performance.

Two other HTMs have improved hardware support for transactions of unbounded size. OneTM (Blundell *et al*., 2007) supports unbounded transaction sizes with simple hardware, but restricts the TM system to execute only one overflowed transaction at a time. Per-block metadata are used to track the read-sets and write-sets for transactions overstepping hardware caches. A special TM-state victim cache is used on transactional data eviction to minimize serialization for overflow transactions. But it may be a bottleneck as transactions scale up. Unlike OneTM, TokenTM (Bobba *et al*., 2008) supports executing multiple overflowed transactions at the same time. To accomplish this, TokenTM associates tokens with each memory block to precisely tracking conflicts on an unbounded number of memory blocks with relatively simple hardware. In both OneTM and TokenTM, non-overflowed transactions are not affected by overflowed transactions.

Nesting: To facilitate software composition, HTMs must allow transactional nesting: starting and ending one transaction from inside another (Larus and Rajwar, 2006). The simplest way to support transactional nesting is the flattening model, which includes all nested transactions in the outmost transaction such as TCC, UTM, LogTM and OneTM. That is all involved transactions share one read-set and one write-set. Unfortunately with flat nesting, a conflict with an inner transaction forces a complete abort of all its ancestors as well. To solve this problem, researchers have developed two optimizations over flat nesting: closed nesting with partial aborts and open nesting.

Closed nesting seeks to improve performance over flattening by aborting and re-executing only the conflicted transaction such as Bulk, Nested LogTM and LogTM-SE. Moss (Moss and Hosking, 2006) and Yossi (Lev and Maessen, 2008) also use this method. It allows each nested transaction to have its own read-sets and write-sets, so that when an inner transaction commits, its read-sets and write-sets merge with the read-sets and write-sets of the next level out. In case of abort, the innermost conflicting transaction rolls back to its original states but not to the top level.

Open nesting can unleash more concurrency than closed nesting such as LogTM-SE and other research by Chung *et al*. (2006), Lev and Maessen (2008), McDonald *et al*. (2006) and Moss and Hosking (2006). Open nesting relaxes the atomicity and isolation guarantees of closed transactions. When an open nested transaction commits, its write set becomes visible to all other transactions, so other transactions can see its updates before the outer transaction has committed and work with them sooner. This may explore more concurrency when shared resources are simultaneously accessed by several large transactions.

Before committing, open nesting acts like closed nesting. When the inner transaction commits, the inner transaction of closed nesting merges its read-sets and write-sets with the outer transaction, while the inner transaction of open nesting clears its read-sets and write-sets and exposes its updates to all threads.

EXCLUSIVE TOPIC FOR STM

Synchronization: Synchronization is the mechanism to guarantee that a transaction attempting to access a logical data will finally finish its work. In general, there are two types of concurrent control: Blocking Synchronization (BS) and Nonblocking Synchronization (NS).

BS is a conventional form of synchronization, constructed with locks, monitors or semaphores. In order to keep consistency, BS forces multiple threads to access critical sections exclusively. Once a critical section is protected by a lock, only the lock owner can access it. Other threads that attempt to acquire the same lock must shift into wait-state until the lock is released by its owner. This wait-state easily leads to severe problems such as deadlock, priority inversion and convoying. In contrast, NS which is used in OSTM, SXM, WSTM, DSTM, ASTM, RSTM and Ananian STM (Ananian and Rinard, 2005; Fraser, 2004; Guerraoui *et al*., 2005a, b; Harris and Fraser, 2003; Herlihy *et al*., 2003a, b; Marathe *et al*., 2005, 2006) can prevent concurrent threads from this wait-state. With NS, a concurrent thread may either abort its transaction, or abort the transaction of conflicting thread.

NS has been classified into three main categories based on their assurances for forward progress: Wait-freedom, Lock-freedom and Obstruction-freedom.

Wait-freedom (Herlihy, 1991) is the strongest assurance to guarantee that all threads contending for concurrent logical data make progress in a bounded number of their own time steps. This feature avoids the occurrence of deadlocks and starvation.

Lock-freedom (Fraser, 2004) is a weaker assurance to guarantee that at least one of the threads contending for concurrent logical data makes progress in a bounded time steps of any of the concurrent threads. This feature avoids the occurrence of deadlocks but not starvation. Obstruction-freedom (Herlihy *et al*., 2003a) is the weakest assurance to guarantees that a thread makes progress in a bounded number of its own steps in the absence of contention. This feature avoids the occurrence of deadlocks but not livelocks (Cheng *et al*., 2007). The problem of livelock can be effectively minimized with simple methods like exponential backoff, or other contention management (Herlihy *et al*., 2003a). In practice, most STMs with NS belong to Obstruction-freedom, as they get freedom from wait-state of concurrent threads while have relative lower costs for implementation than Lock-freedom and wait-freedom.

Early researchers for STM systems focus on nonblocking data structures with NS to guarantee forward progress. Many researchers in recent STM systems such as TL, Ennals STM, Bartok STM, AUTOLOCKER, Haskell STM and McRT-STM (Dice and Shavit, 2006; Ennals, 2006; Harris *et al*., 2006; McCloskey *et al*., 2006; Perfumo *et al*., 2008; Saha *et al*., 2006) suggest that NS is more complex and lower performing than BS, if NS is combined with LVM and BS is combined with EVM. Furthermore, NS may cause more memory traffic than BS. Therefore, they suggest that in order to get higher performance as well as to have forward progress assurances, a STM can use BS with timeouts to find and resolve deadlocked transactions in the implementation aspect. While in the logic aspect, it provides users with transactions (a nonblocking abstraction).

CHALLENGES FOR HTM

Although HTM provides an efficient solution to ease parallel programming, it poses several challenges to designers. Two most serious challenges are I/O and ISA.

Input/output: The relationship between Input/Output (I/O) operations and transactions is a significant research challenge. One serious problem is that a transaction that executed an I/O operation may need to roll back at a conflict. In some cases, I/O consists of interactions with the world outside of the HTM system. If a transaction aborts, its I/O operations should roll back too, which may be difficult or impossible in general. Some rollbacks my accomplished by buffering the read and write set of a transaction, but it may not work even in simple situations, such as a transaction that is waiting for user input. A more general approach is to indicate a single privileged transaction and confirm its completion, by ensuring it succeeds over all conflicting transactions. A more general approach is to designate a single privileged transaction that runs to completion, by ensuring it triumphs over all conflicting transactions. Only the privileged transaction can perform I/O and the privilege can be passed between transactions. Unfortunately, it limits the amount of I/O a program can perform.

Instruction set architecture support: Because no practical HTM systems have been used yet, no standard TM supported Instruction Set Architecture (ISA) exist. The ISA extension suggestions have been proposed range from no ISA support in Implicit Transactions to recent detailed support mechanism in UTM. In HMTM, ISA support is trivial. Even the start transaction instruction is unnecessary in their system. But many complex models can be supported or emulated by their system. In contrast, McDonald *et al.* (2006) provide explicit ISA support for two-phase transaction commit, closed and open nested transactions and support handlers for transaction commit, conflict and abort. Although, many researches on TM supported ISA have been carried out and have made some progress, to provide the right level of ISA support which can be widely adopted in practice for TM is still a challenge.

CHALLENGE FOR STM

Although STM provides an efficient solution to ease parallel programming, it brings several challenges to designers. The most serious challenge is isolation in transactions. The isolation of TM is defined as: how nontransaction code and transaction code share data. Currently, there is strong isolation and weak isolation for TM. If the systems use strong isolation, the nontransaction code cannot read or write uncommitted TM data. They guarantee the semantics correctness, but degrade the performance of STM systems. If the systems use weak isolation, isolation is only used between transactions and the nontransaction code can read or write uncommitted TM data. They have high performance but may get non-determined results. Up to now, most STM systems use weak isolation model. The future research will focus on how to trade off between performance and semantics correctness.

CONCLUSION

Transactional memory provides a flexible and easy mechanism for parallel programming in multicore processors. This paper firstly presented three key research areas for all implementation schemes according the following aspects: version management, conflict detection and contention management. Then we descript their exclusive research hot topic from different implementation schemes. From the analysis above, we get a research trend that the future transactional memory systems will be more likely combined with flexibility of software implementation and high performance of hardware implementation, the hybrid will become the next hot research field. Supporting different ISA, strong atomicity and I/O is still the subject of active research.

REFERENCES

Ananian, C.S. and M. Rinard, 2005. Efficient object-based software transactions. Proceedings of Workshop on Synchronization and Concurrency in Object-Oriented Languages, Oct. 16, San Diego, California, USA., pp: 1-10.

Ananian, C.S., K. Asanovic, B.C. Kuszmaul, C.E. Leiserson and S. Lie, 2005. Unbounded transactional memory. Proceedings of the 11th International Symposium on High-Performance Computer Architecture, Feb. 12-16, San Francisco, California, pp: 316-327.

Blundell, C., E.C. Lewis and M.M.K. Martin, 2005. Deconstructing transactional semantics: The subtleties of atomicity. Proceedings of the Workshop on Duplicating, Deconstructing and Debunking, June 2005, Madison, Wisconsin USA., pp: 12-18.

Blundell, C., J. Devietti, E.C. Lewis and M.M.K. Martin, 2007. Making the fast case common and the uncommon case simple in unbounded transactional memory. ACM SIGARCH Comput. Archit. News, 35: 24-34.

Bobba, J., N. Goyal, M.D. Hill, M.M. Swift and D.A. Wood, 2008. TokenTM: Efficient execution of large transactions with hardware transactional memory. Proceedings of the International Symposium on Computer Architecture, Jun. 21-25, Beijing, China, pp: 127-138.

Ceze, L., J. Tuck, J. Torrellas and C. Cascaval, 2006. Bulk disambiguation of speculative threads in multiprocessors. ACM SIGARCH Comput. Archit. News, 34: 227-238.

Cheng, X. and H. Liu *et al*., 2007. A fault tolerance deadlock detection/resolution algorithm for the and-or model. J. Comput. Res. Dev., 44: 798-805.

Chuang, W., S. Narayanasamy, G. Venkatesh, J. Sampson and M. Van Biesbrouck *et al*., 2006. Unbounded page-based transactional memory. Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 21-25, San Jose, California, USA., pp: 347-358.

Chung, J., C.C. Minh, A. McDonald, T. Skare and H. Chafi 2006. Tradeoffs in transactional memory virtualization. Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. Oct. 21-25, San Jose, California, USA., pp: 371-381.

Dice, D. and N. Shavit, 2006. What Really Makes Transactions Faster?. Proceedings of the 1st ACM SIGPLAN Workshop on Transactional Computing, Jun. 1, ACM Press, Ottawa, Canada, pp: 1-11.

Dice, D. and N. Shavit, 2007. Understanding tradeoffs in software transactional memory. Proceedings of the International Symposium on Code Generation and Optimization, Mar. 11-14, San Jose, CA., pp: 21-33.

Ennals, R., 2006. Software transactional memory should not be obstruction-free. Intel Research Cambridge Tech Report. Report No. IRC-TR-06-052. http://www.cs.wisc.edu/trans-memory/misc-papers/052_Rob_Ennals.pdf.

Fraser, K., 2004. Practical lock-freedom. University of Cambridge, Report No. UCAM-CL-TR-579. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

Fu, C., Z. Wu, X. Wang and X. Yang, 2009. A review of software transactional memory in multicore processors. Inform. Technol. J., 8: 1269-1274.

Guerraoui, R., H. Maurice and P. Bastian, 2005a. Polymorphic contention management. Proceedings of the 19th International Symposium on Distributed Computing, Sept. 26-29, Springer, Cracow, Poland, pp: 26-29.

Guerraoui, R., M. Herlihy and B. Pochon, 2005b. Toward a theory of transactional contention managers. Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing. Las Vegas, NV, USA., Jul. 17-20, ACM Press, New York, pp: 258-264.

Hammond, L., V. Wong, C. Mike, D.C. Brian and D.D. John *et al*., 2004. Transactional memory coherence and consistency. Proceedings of the 31st Annual International Symposium on Computer Architecture, Jun. 19-23, IEEE Press, Munich, Germany, pp: 102-113.

Harris, T. and K. Fraser, 2003. Language support for lightweight transactions. Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages and Applications, Oct. 26-30, Anaheim, California, USA., pp: 388-402.

Harris, T., M. Plesko, A. Shinnar and D. Tarditi, 2006. Optimizing memory transactions. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Jun. 11-14, Ottawa, Ontario, Canada, pp: 14-25.

Harris, T., S. Marlow, H. Maurice and S. Peyton-Jones, 2005. Composable memory transactions. Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Jun. 15-17, ACM Press, Chicago, IL, USA., pp: 48-60.

Herlihy, M. and J.E.B. Moss, 1993. Transactional memory: Architectural support for lock-free data structures. Proceedings of the 20th Annual International Symposium on Computer Architecture, May 16-19, San Diego, CA, USA., pp: 289-300.

Herlihy, M., 1991. Wait-free synchronization. ACM Transact. Programming Languages Syst., 13: 124-149.

Herlihy, M., V. Luchangco and M. Moir, 2003a. Obstruction-free synchronization: Double-ended queues as an example. Proceedings of the 23rd International Conference on Distributed Computing Systems, May 19-22, IEEE Press, Providence, Rhode Island, USA., pp: 522-529.

Herlihy, M., V. Luchangco, M. Moir and W.N. Scherer, 2003b. Software transactional memory for dynamic-sized data structures. Proceedings of the 22th Annual Symposium on Principles of Distributed Computing, Jul. 13-16, Boston, Massachusetts, USA., pp: 92-101.

Hicks, M., J.S. Foster and P. Polyvios, 2006. Lock inference for atomic sections. Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers and Hardware Support for Transactional Computing, Jun. 11, Ottawa, Canada, pp: 1-8.

Larus, J.R. and R. Rajwar, 2006. Transactional Memory. Morgan and Claypool Publishers, Baltimore City, ISBN: 1598291246.

Lev, Y. and J.W. Maessen, 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, Feb. 20-23, Salt Lake City, UT, USA., pp: 197-206.

Li, K. and X. Yang, 2000. Improving the performance of checkpointing scheme with task duplication. Chinese J. Electr., 28: 33-35.

Marathe, V.J., W.N. Schere-III and S.H. Michael, 2005. Adaptive software transactional memory. Proceedings of the 19th International Symposium on Distributed Computing, Sept. 26-29, Cracow, Poland, pp: 354-368.

Marathe, V.J., M.F. Spear, H. Christopher, A. Athul, E. David, N.S. William III and L.S. Michael, 2006. Lowering the overhead of nonblocking software transactional memory. Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, Jun. 11, Ottawa, Canada, pp: 249-282.

McCloskey B., F. Zhou, G. David and B. Eric, 2006. Autolocker: Synchronization inference for atomic sections. Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Jan. 11-13, Charleston, South Carolina, USA, ACM Press, pp: 346-358.

McDonald, A., J. Chung, B.D. Carlstrom, C.C. Minh, H. Chafi, C. Kozyrakis and K. Olukotun, 2006. Architectural semantics for practical transactional memory. ACM SIGARCH Comput. Archit. News, 34: 53-65.

Moore, K.E., J. Bobba, M.J. Moravan, M.D. Hill and D.A. Wood, 2006. LogTM: Log-based transactional memory. Proceedings of the 12th International Symposium on High-Performance Computer Architecture, Feb. 11-15, Austin, Texas, USA., pp: 254-265.

Moravan, M.J., J. Bobba, K.E. Moore L. Yen and M.D. Hill *et al*., 2006. Supporting nested transactional memory in logTM. Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 21-25, San Jose, California, USA., pp: 359-370.

Moss, J.E.B. and A.L. Hosking, 2006. Nested transactional memory: Model and architecture sketches. Sci. Comput. Program., 63: 186-201.

Perfumo C., N. Sonmez, S. Srdjan, U. Osman, C. Adrian, V. Mateo and H. Tim, 2008. The Limits of Software Transactional Memory (STM): Dissecting haskell STM applications on a many-core environment. Proceedings of the 5th Conference on Computing Frontiers, May 5-7, ACM Press, Ischia, Italy, pp: 67-78.

Plakal, M., D.J. Sorin, A.E. Condon and M.D. Hill, 1998. Lamport clocks: Verifying a directory cache-coherence protocol. Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures. Puerto Vallarta, Mexico, Jun. 28-Jul. 02, ACM Press, New York pp: 67-76.

Rajwar, R. and J.R. Goodman, 2002. Transactional lock-free execution of lock-based programs. Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 5-9, ACM New York, USA, pp: 5-17.

Rajwar, R., M. Herlihy and K. Lai, 2005. Virtualizing transactional memory. Proceedings of the 2nd Annual International Symposium on Computer Architecture, Jun. 04-08, Madison, Wisconsin USA., pp: 494-505.

Saha, B., A.R. Adl-Tabatabai, R.L. Hudson, C.C. Minh and B. Hertzberg, 2006. McRT-STM: A high performance software transactional memory system for a multi-core runtime. Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Mar. 29-31, New York, USA., pp: 187-197.

Schere III, W.N. and M. Scott, 2004. Contention management in dynamic software transactional memory. Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs, Jul. 25-26, St John's, Newfoundland, Canada, pp: 1-10.

Scherer III, W.N. and M.L. Scott, 2005. Advanced contention management for dynamic software transactional memory. Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing, Las Vegas, NV, USA., Jul. 17-20, ACM New York, USA., pp: 240-248.

Shavit, N. and D. Touitou, 1995. Software transactional memory. Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, Aug. 20-23, Ottowa, Ontario, Canada, pp: 204-213.

Shpeisman, T., V. Menon, A.R. Adl-Tabatabai, S. Balensiefer and D. Grossman *et al*., 2007. Enforcing isolation and ordering in STM. Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, Jun. 10-13, San Diego, California, USA., pp: 78-88.

Sorin, D.J., M. Plakal, A.E. Hill, M.M.K. Martin, D.A. Wood, 2002. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. IEEE Trans. Parallel Distrib. Syst., 13: 556-578.

Tabatabai, A.A., B.T. Lewis, V. Menon, B.R. Murphy, B. Saha and T. Shpeisman, 2006. Compiler and runtime support for efficient software transactional memory. Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, Jun. 11-14, Ottawa, Ontario, Canada, pp: 26-37.

Wang, K.F., Z.Z. Ji and M.Z. Hu, 2005. Path-based next N trace prefetch in trace processors. Microprocess. Microsyst., 29: 273-288.

Wang, K.F., Z.Z. Ji and M.Z. Hu, 2006a. Boosting SMT trace processors performance with data cache misssensitive thread scheduling mechanism. Microprocessors Microsyst., 30: 225-233.

Wang, K.F., Z.Z. Ji and M.Z. Hu, 2006b. Simultaneous multi threading trace processors: Improving trace processors performance. Microprocessors Microcyst., 30: 102-116.

Wang, X., Z. Ji, C. Fu and M. Hu, 2009. A review of hardware transactional memory in multicore processors. Inform. Technol. J., 8: 965-970.

Yen, L., J. Bobba, M.R. Marty, K.E. Moore and H. Volos *et al*., 2007. LogTM-SE: Decoupling hardware transactional memory from caches. Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture, Feb. 10-14, Scottsdale, AZ, USA., pp: 261-272.

Zilles, C. and R. Rajwar, 2007. Brief announcement: Transactional memory and the birthday paradox. Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures, San Diego, California, USA., Jun. 09-11, ACM Press, New York, pp: 303-304.