http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 8 (7): 965-970, 2009
ISSN 1812-5638

A Review of Hardware Transactional Memory in Multicore Processors

X. Wang, Zhenzhou Ji, Chen Fu and Mingzeng Hu
Post Box 1209, No.13 Fa Yuan Street, Harbin Institute of Technology, 150001, China

Abstract: In this study, we give a review of the current Hardware Transactional Memory (HTM) systems for Multicore processors. Hardware transactional memory systems are classified into the following three categories: how to perform version management and conflict detection, whether to support unbounded transactional memory and whether to support transactions nesting. Finally, we discussed two active research challenges: the relationship between transactional memory and Input/Output operations and Instruction Set Architecture (ISA) supporting.

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

INTRODUCTION

Multicore processors are now popular 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 parallel programming (Herlihy and Moss, 1993).

Transactional Memory (TM) tries to replace critical sections protected by locks in a multi-threaded parallel program by transactions (Minh *et al*., 2007). Compared to critical sections, transactions have several advantages. First, programmers are liberated from guaranteeing 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 guarantee transactions either commit or abort, consistency to guarantee transactions use the same total order during the whole process and Isolation to guarantee that each transaction's operations are isolated to other transactions.

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 aborts and restarts. As a transaction starts, it checkpoints registers to save old values, which can be restored in case of aborting. 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 appears between two transactions, one of them rolls back according to register checkpoint.

Transactional memory can be implemented in hardware, software, or a hybrid of the two. Software Transactional Memory (STM) systems (Harris and Fraser, 2003; Herlihy *et al*., 2003; Larus and Rajwar, 2006; Saha *et al*., 2006; Shavit and Touitou, 1995) are easy to implement and require 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, compared with strong atomicity, weak atomicity will make more writing and debugging difficulties, reducing the benefits 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

Corresponding Author: Xiaoqun Wang, Post Box 1209, No.13 Fa Yuan Street, Harbin Institute of Technology, 150001, China
Tel: +86-451-86413354, +86-13704804719

transaction and leverage the cache coherence protocol to detect conflicts between concurrent transactions (Lance *et al*., 2004; Moore *et al*., 2006; Yen *et al*., 2007). By using hardware, HTM systems cut down the overhead of fine-grained locks and they have higher speed not only than STMs, but also than lock-based synchronization for many cases. In addition, they can automatically check every memory references of all the active transactions under the help of the cache coherence protocols. Thus, they ensure strong atomicity with no or minor additional overhead.

A HTM SYSTEM FRAME WORK

Hardware Transactional Memory (HTM) systems use hardware to perform basic TM operations for reducing their overhead. They also make parallel programming easier without caring about low level mechanisms. Most HTMs have similar structure. They have two basic functions: version management and conflict detection, both of which are often implemented by enhancing processor caches.

First, hardware must provide version management by storing both the new data and old data of a transaction. If a transaction commits, the new data will be published to others. If a transaction aborts, the old data will be kept.

Second, hardware must provide conflict detection among transactions by recording the read-set (addresses read) and write-set (addresses written) of a transaction. A conflict occurs when an overlap appears in the write-set of two transactions or the write-set of one and the read-set of another. Most HTMs augment cache coherence protocols to detect conflict.

Figure 1 shows a framework of HTM system. Transactional memory registers are used to store transaction states such as transaction id, nesting level, read-set and write-set etc. Register checkpoints are stored in extra register files. When a transaction is aborted, it will roll back to its original register checkpoint. Cache lines are added read and write bits to signal transactional memory references. Conflict detection is combined with cache coherence protocols, where every memory request from other processors will be checked for conflict. Version management will wake up when transactions commit, abort and conflict.

HTM SYSTEM TAXONOMY

Version management and conflict detection: Eager or lazy: Besides conflict detection and version management, a practical HTM system always has conflict resolution function, in order to make it more efficient. According to former proposed HTM classification method (Bobba *et al*., 2007), each function represents a major choice in the HTM design space.

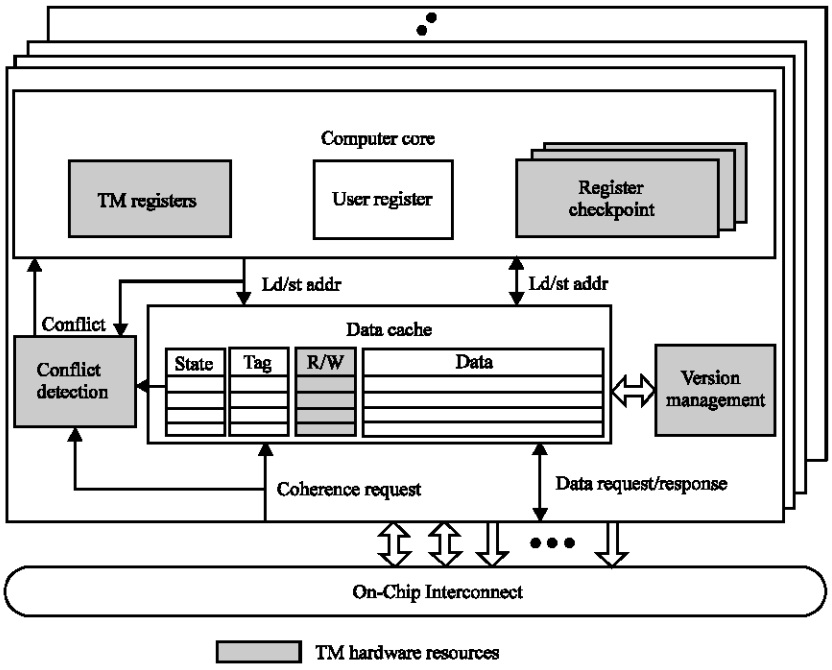

Fig. 1: A framework of hardware transactional memory system

Table 1: A transactional memory taxonomy

Conflict detection	Version management	Conflict resolution	HTM systems
Lazy	Lazy	Committer wins	TCC (Lance *et al*., 2004), Bulk (Ceze *et al*., 2006)
Eager	Lazy	Requester wins	LTM (Ananian *et al*., 2005)
Eager	Eager	Requester stalls with conservative deadlock avoidance	UTM (Ananian *et al*., 2005), LogTM (Moore *et al*., 2006), Nested LogTM (Moravan *et al*., 2006), LogTM-SE (Yen *et al*., 2007),

The first design choice is when to detect conflicts in read-sets and write-sets. There are two methods: eager conflict detection and lazy conflict detection. Using eager conflict detection, a HTM system detects conflicts when a transactional thread wants to refer memory. While, using lazy conflict detection, an HTM system detects conflicts when the transactions commit. To improve performance, eager conflict detection may deal with some conflicts using stalls, rather than aborts, because no transaction can see an uncommitted or stale value. By contrast, lazy conflict detection allows an implementation to batch conflict checking (Ceze *et al*., 2006; Lance *et al*., 2004).

The second design choice belongs, how to store new data and old data. Lazy version management puts old values in memory for making fast aborts. Conversely, eager version management stores newly values in target address for making fast commits and slow aborts. But it may exacerbate the effects of contention.

The third design choice is what to do when a conflict appears. The resolution policy has three choices: committer win, requester win and requester stall with conservative deadlock avoidance.

According to the three design choices, previously proposed HTM systems fall into three groups which are illustrated in Table 1.

The first group of HTM system is Lazy conflict detection, Lazy version management and Committer Wins (LLCW). Lazy conflict detection, Lazy version management and Committer Wins systems, such as TCC (Lance *et al*., 2004) and Bulk (Ceze *et al*., 2006) cannot write new values in target address until a transaction commits. To achieve a global serial commit order, a completing transaction arbitrates for a commit token or bus. It then commits and informs other transactions of its write-set and exposes its updates. If a memory block in the committing transaction's write-set is also in another transaction's read-set, the HTM detects a conflict and then aborts the reader's transaction. Two benefits can be brought by this policy. First, at least a transaction is guaranteed to commit even if other transactions abort, thus program forwarding is ensured. Second, because other transactions will be aborted, the committing transaction is never delayed.

The second group of HTM system is Eager conflict detection, Lazy version management and Requester Wins (ELRW). Eager conflict detection, Lazy version management and Requester Wins systems, such as LTM (Ananian *et al*., 2005), detect conflicts on every private memory references, but will not update until commit. When conflicting request happens, the requester always forwards and the conflicting non-request transactions must abort. Like LLCW systems, aborts are simplified in the ELRW systems because old values store in target address until commit. Like LTM, VTM (Rajwar *et al*., 2005) also combines lazy version management with eager conflict detection, but none determined conflict resolution methods are given in that study.

The third group of HTM system is Eager conflict detection, Eager version management and Requester Stalls with conservative deadlock avoidance (EERS). EERS systems are described in the research literatures of UTM (Ananian *et al*., 2005), LogTM (Moore *et al*., 2006), Nested LogTM (Moravan *et al*., 2006) and LogTM-SE (Yen *et al*., 2007). Those systems also detect conflicts on every private memory references, but complete updates in target address and stores old values to a per-thread log.

Eager conflict detection, Lazy version management and Requester Wins deals with conflicts by stalling the requester or aborting only when a stall would lead to a potential deadlock. Some EERS systems use timestamp to detect potential deadlocks. For an example, when a transaction that has stalled an older transaction would itself stall on an older transaction. Commits are simplified with Eager version management policy. In an opposite position, aborts are slowed because the log needs to be processed.

Whether supporting unbounded TM: 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. By modifying the coherence protocol, LogTM allows transactional threads to migrate their states from the cache.

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, cache blocks accessed by transactions are put into XADT structure. For further optimization, VTM brings additional 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 degrades performance by unnecessarily serializing non-conflicting large transactions increase the probability of false conflicts, leading to a further worsening of performance.

Two other HTMs support for unbounded transactions with additional hardware. 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.

Whether supporting nesting: To develop composable software, 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 can improve performance by aborting and re-executing only the conflicted transactionsuchas 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 have more concurrency than closed nesting such as LogTM-SE and other works (Chung *et al*., 2006; Lev and Maessen, 2008; McDonald *et al*., 2006; Moss and Hosking, 2006). Open nesting relaxes the atomicity and isolation guarantees of closed transactions. When an open nested transaction commits, all the other transactions can see its updates immediately and continue their work with the new data earlier, without delaying until the outer transaction commits. 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.

CHALLENGES

Although, HTM provides an efficient solution to ease parallel programming, it brings 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 difficult 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, causing difficulties to complete the task. 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 general method is to point out a single dedicated transaction and confirm its completion, by ensuring it succeeds over all conflicting transactions. Only that dedicated transaction can perform I/O and its special right can be delivered between transactions. However, this method limits the total I/O numbers of a process.

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 (Vallejo *et al*., 2005) to elaborate support in UTM. In HMTM, ISA support is so trivial that the start transaction instruction is not required in their system. But many complex models can be supported or emulated by their system. In contrast, some researchers provide explicit ISA support such as two-phase transaction commit, abort, closed and open nesting etc. (McDonald *et al*., 2006). 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.

CONCLUSION

Hardware Transactional memory provides an efficient and easy mechanism for parallel programming in multicore processors. This study presented three categories for hardware transactional memory systems: how to detect conflict and resolve conflict, whether to support unbounded transactional memory and transactions nesting. We also analyze two active research challenges: the relationship between transactional memory and Input/Output operations and Instruction Set Architecture supporting. If these difficulties can be resolved, TM will become an important model of parallel programming.

REFERENCES

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: 1-7.

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., K.E. Moore, H. Volos, L. Yen, M.D. Hill, M.M. Swift and D.A. Wood, 2007. Performance pathologies in hardware transactional memory. ACM SIGARCH Comput. Archit. News, 35: 81-91.

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.

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, 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.

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 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, Jun. 11-14, Ottawa, Ontario, Canada, pp: 14-25.

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., V. Luchangco, M. Moir and W.N. Scherer, 2003. 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.

Lance, H., W. Vicky, M. Chen, B.D. Carlstrom and J.D. Davis *et al*., 2004. Transactional memory coherence and consistency. ACM SIGARCH Comput. Archit. News, 32: 102-113.

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

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.

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.

Minh, C.C., M. Trautmann, J. Chung, A. McDonald and N. Bronson *et al*., 2007. An effective hybrid transactional memory system with strong isolation guarantees. Proceedings of the 34th Annual International Symposium on Computer Architecture, Jun. 09-13, San Diego, California, USA., pp: 69-80.

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.

Rajwar, R., M. Herlihy and K. Lai, 2005. Virtualizing transactional memory. Proceedings of the 32nd 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.

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.

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.

Vallejo, E., M. Galluzzi, A. Cristal, F. Vallejo and R. Beivide, 2005. Implementing kilo-instruction multiprocessors. Proceedings of the International Conference on Pervasive Services, Jul. 11-14, Santorini, Greece, pp: 325-336.

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, Jun. 09-11, San Diego, California, USA., pp: 303-304.