Conflict Graph Based Hardware Transactional Memory

Kun Zeng, National University of Defense Technology
National Laboratory for Parallel and Distributed Processing, School of Computer
National University of Defense Technology
ChangSha, Hunan, China
Email: kunzeng.nudt@gmail.com

***Abstract*—This paper proposes a novel transactional memory design: conflict graph based hardware transactional memory. It allows two conflicting transactions both to commit if they do not violate the condition of serializability. Simulation results show that conflict graph based hardware transactional memory outperforms the state-of-art transactional memory system.**

Keywords-transactional memory; conflict detection; serializability; conflict graph

I. INTRODUCTION

As the development of chip-multiprocessing, parallel programming is becoming more and more important. Transactional memory [1][2] is a promising technique to ease parallel programming by simplifying the concurrency control of accesses to shared memory locations.

Transactional memory encapsulates a number of memory accesses into a transaction which ensures atomicity, isolation and serializability. Serializability ensures the result of concurrent execution of transactions is the same with a sequential execution of the transactions in a certain order. During the execution of a transaction, the read/write set is recorded. If two transactions access the same memory location and at least one of them is a write access, a conflict will be detected and one of the conflicting transactions are aborted. This conflict handling strategy can ensure the serializability of the concurrent execution of transactions, but it limits the concurrency too much. As is shown in Fig. 1 a), two transactions, T0 and T1 access the variable A and T1 is a write. When T1 commits, T0 should be aborted. But even if T0 commits successfully,the serializability of the concurrent execution of T0 and T1 is still maintained. Fig. 1 b) shows a possiable serial execution order.

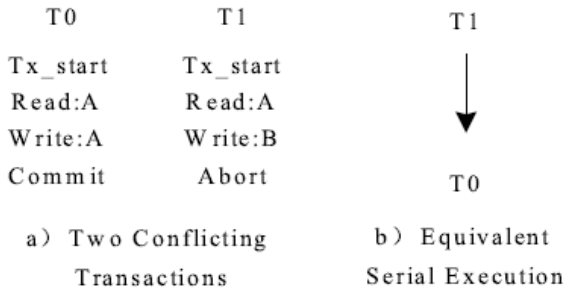

Figure 1. A Serializable Execution of Two Transactions with Conflict

This paper proposes CGHTM (conflict graph based hardware transactional memory), which allows two conflicting transactions both to commit. We test CGHTM against EasyTM [7], which extends the directory cache protocol to support eager conflict detection and lazy version management. Simulation results show that CGHTM reduces transaction aborts significantly, which leads to better performance.

II. SERIALIZABILITY AND CONFLICT GRAPH

A transaction can be denoted as a number of operations in a certain order. The operations allowed are read, write and commit. The read operation operates directly to the shared data while the write operation does not. When a write operation is performed to a memory location, the new value is not directly written to it. The new value is buffered in a private buffer until the commit of the transaction.

A schedule of a set of transactions is a sequence of the actions performed by the transactions in a certain order. A conflict or dependence exists between two actions from two different transactions if they access the same address and at least one of them is a write. A conflict can only happens between a read operation and a commit operation or two commit operations since all the write operation does not take effect until transaction commit. A schedule is a serial schedule if the actions from different transactions do not interleave. Two schedules are conflict equivalent if they contain the same actions and the order of the conflicting actions is the same. If a schedule is conflict equivalent to a serial schedule, then we say it is a conflict serializable schedule [4].

The serializability of a schedule can be judged through the conflict graph [3][4]. The conflict graph of a schedule is a directed graph, of which the nodes represent the transactions. Conflicting transactions are connected with an edge, which come out of the transaction that performs the conflicting operation earlier and go into the transaction that performs the conflicting operation later in time. A schedule is serializable if its conflict graph is acyclic [3][4].

As is shown in Fig. 2 a), The transaction T0 and T3 both access variable C and the operation from T3 is a write. In most proposed hardware transactional memory designs, the commit of T3 will abort T0. The commit of T1 and T2 will also cause the abort of T0. However, even if the commits of T1, T2 and

978-1-4244-5539-3/10/$26.00 ©2010 IEEE

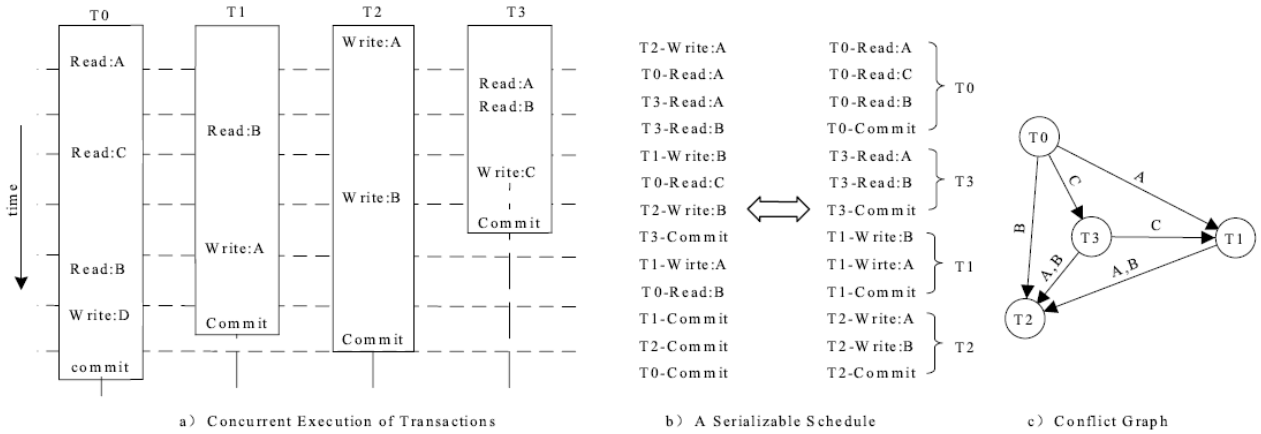

Figure 2. An Example of Concurrent Execution of Transactions

T3 happen before the commit of T0, the conflict graph of the schedule is still acyclic (2 c)). The corresponding serial schedule is shown in Fig. 2 b).

Most of the state-of-art hardware transactional memory systems do not allow conflicting transactions both to commit even if the conflict graph is acyclic. This limits the concurrency of the execution of transactional programs and degrades performance. Therefore, we propose the CGHTM (conflict graph based hardware transactional memory), which directly use conflict graph as the key mechanism for concurrency control.CGHTM receives all the schedules of which the conflict graph is acyclic.

III. DESIGN AND IMPLEMENTATION

A. Maintainance of the Conflict Graph

Each processor should maintain three sets to maintain the conflict graph dynamically: the predecessor set (PS), successor set(SS) and write-write conflict set(WS). If processor Pi runs transaction Ti, then the PS contains all the processor that runs the transactions which are predecessors of Ti in the conflict graph. If transaction Tj, which is running on processor Pj, issues a read operation to the shared variable written by Ti, then Pj should be added to the PS of Pi. The SS of Pi contains all the processor that runs the transactions which are successors of Ti in the conflict graph. If transaction Tj, which is running on processor Pj, issues a write operation to the shared variable read by Ti, then Pj should be added to the SS of Pi. The WS of Pi contans all the processors that runs the transactions which have issued a write operation to the shared variables written by Ti. When a write-wirte conflcit is deteced between transaction Ti and Tj, the direction of the edge between Ti and Tj can not be determined until one of them commits. So we should keep the processors which runs transactions with write-write conflict in the WS until one of them commits.

The three set are implemented as three bitvectors, of which each bit represnets one processor in the system. The three bit vectors are dynamically updated according to the conflict detection results. When a transaction is ready to commit, it just detects whether there is a circle in the conflcit graph to determine if it should abort.

B. Conflict Detection

We extends the directory based cache coherence protocol to detect the conflcits. The directory should responds to two extra messages: tx_read and tx_write. Moreover, the directory should records the readers and wrters for all the cache blocks.

When a transaction issues a read/write operation to a certain cache block, it should send a tx_read/tx_write message to the directory. Then the directory should respond with a tx_ ack message. tx_ack message tells the processor which processor have written/read the cache block. The processor that receives inform message updates its PS, SS and WS, and responds with inform _ack message.

Fig. 3 shows an example of conflict detection. Two transactions are running on two processors, P0 and P1 correspondingly. When P1 read cache block A, it sends a tx_ read(A) message to the directory. The directory add P1 to the reader set of the cache block A and responds with tx_ack message. P1 won't send any messages to other processors since T1 is the only reader of A currently. Then P0 is about to read

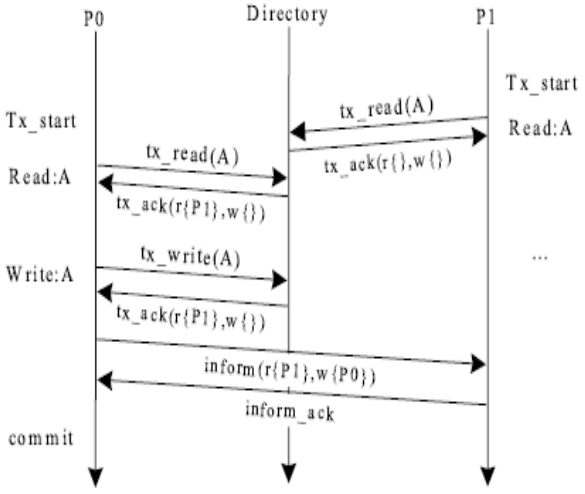

Figure 3. The Conflict Detection Between Processors

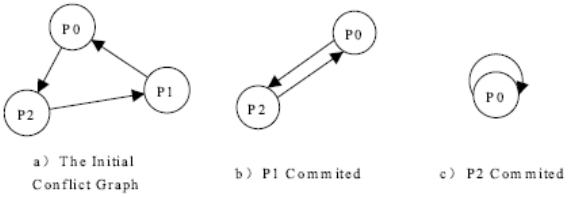

Figure 4. The Circle Detection Process

cache block A. It sends tx_read(A) message to the directory. The directory responds with a tx _ack message and adds P0 to the reader set of A. Until now, both P0 and P1 completes a read operation and no conflict happens. There is no need to communicate with the directory for the successor read operation from P0 and P1 to cache block A.

Then P0 is about to write cache block A. It sends a tx_ write(A) message to the directory. The directory adds P0 the writer list of A and responds with a tx_ ack message, which tells P0 that P1 have read cache block A. This means the commit operation of P1 in the future will cause a conflict with the read operation of P0. P0 adds P1 to its PS and send P1 a inform message to tell P1 about the conflcit detected. When P1 receives the inform message, it adds P0 to its SS and responds with a inform_ ack message. After P0 receives the inform_ ack message, the write operation to A is completed. The successor operation from P0 to A incures no communication to the directory until the commit of operation of P0.

C. Conflict Circle Detection

When a transaction is ready to commit, it should detects whether a circle exists in the conflict graph. The most direct way to detect a circle is to send circle detection messages to its predecessors. Its predecessors then forward the message to their predecessors ... and so on. If the circle detection message returns to the source processor, a circle is detected. The problem with this method is that it can detect the existence of a circle easily but it cannot ensure the non-existence of circles. Moreover the conflict detection messages may consume a lot of network bandwidth. This paper proposes a conflict detection method which incure no explicit circle detection messages between processors.

Fig. 4 shows an example which illustrates the process of circle detection. At the initial state, there is a conflict circle among processor P0, P1 and P2. When P1 is ready to commit, it detects whether a self-circle exists. A self-circle means a transaction ”conflicts” with itself. If no self-circle exists, P1 begin its commit operation. During the commit, P1 sends messages to its predecessors to add new edges to the conflict graph. These edges come out of the successors of P1 and go into the predecessors of P1. After the commit of P1, P2 becomes ready to commit. Also, it detects no self-circle and commit succefully. A new edge is added so that the conflict relationship is forwarded correctly. After the commit of P2, P0 is the only node in the conflict graph and a self-circle exists. P0 detects the self-circle and aborts

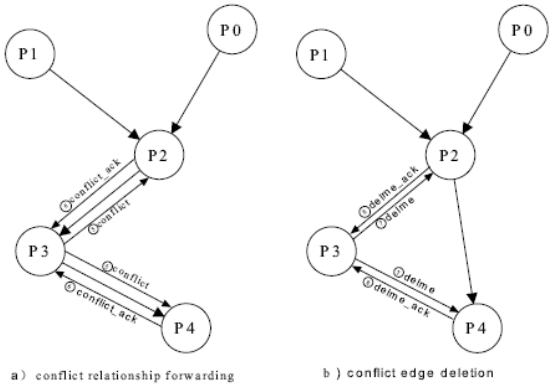

Figure 5. Handling Transaction Commit

D. The Commit/Abort of Transaction

When a transaction is ready to commit, the PS, SS and WS of the corresponding processor contains the information about the conflict relationship between the transaction and other transactions. The commit and abort operation is performed according to the three sets.

The commit process can be divided into two sub-processes: commit validation and commit execution. The commit validation consists of two steps: self-circle detection and conflict relationship forwarding. self-circle detection can be done by checking whether the PS contains itself. conflict relationship forwarding is done by sending the conflict message to its predecessors and successors. Its predecessors will update its SS according to the conflict message and its successors will update its PS according to the depend message. As is shown in Fig. 5 a), processor P3 setup a conflict edge between its predecessor P2 and successor P4. If a processor receives a conflict message, it should respond with a conflict ack message.

The commit execution consists of three steps: data write back, commit acknowledgment and conflict edge deletion. The data write back is similar to a non-transactional write. The processor sends a write request to the directory to get exlusive access to the cache block and then write back dirty data. The data write back will invalidate all other copies of the cache block in other processors. The commit acknowledgment is done by sending delme messages. As is shown in Fig. 5 b), processors that receives delme message delete the message sender from its PS, SS, and WS. After the previous two steps, the processor clear its PS, SS, WS and finishes the commit operation. There are three situations under which a transaction should abort. The first one is commit validation failure. The existence of self-circle and conflict nack message are the main causes for a commit validation failure. The second one is that a system exception comes up during the commit operation. The last one is some other processor non-transactionally modifies a cache block which is accessed by the commiting processor.

The abort of a transaction takes three actions. Firstly, the

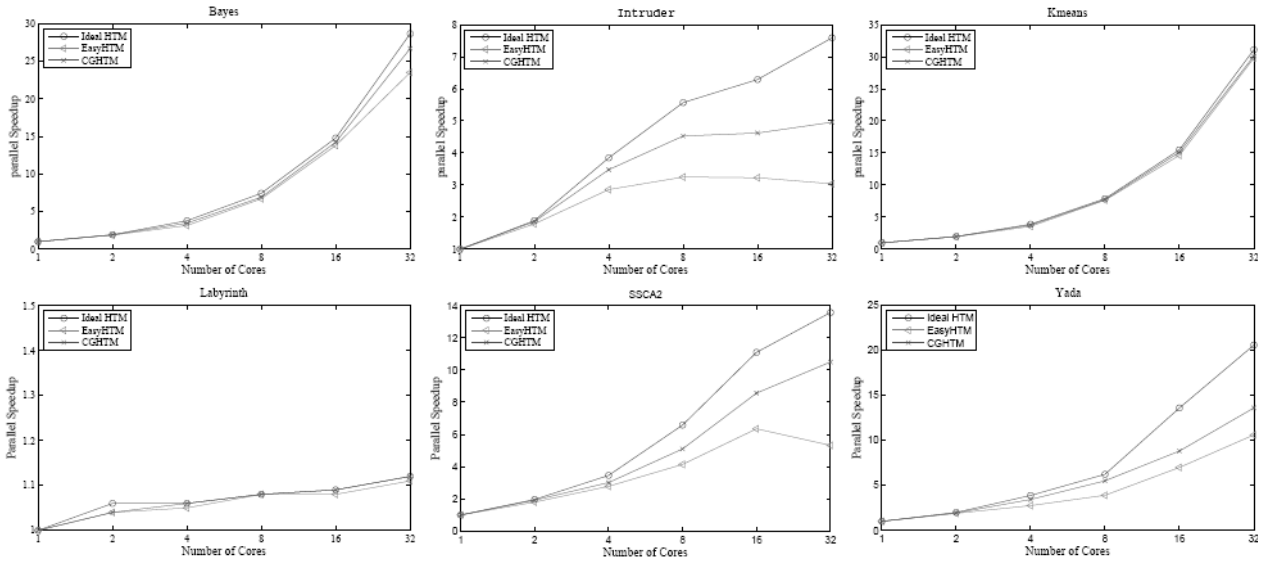

Figure 6. Speedup Achieved for The Kernels

aboring processor should send delme message to the predecessors and successors to delete it self from the conflict graph. Secondly, it discards all the modifications to the cache blocks during the execution of the transaction. Lastly, it clear its PS, SS and WS. After the three actions above, the processor restarts the execution of the transaction.

E. Hardware Cost Estimation

The hardware cost to support conflict grapgh based transactional memory consists of two aspects. The first aspects is the exetention to the processor core. Each processor core should maintain three bit vectors to records the three sets (PS, SS and WS). Also a three-bit transaction status register should be added. The status register contains two fields: an one-bit T (transactional) field used to tag whether the processor is running transactional code, an two-bit S (State) field tags the current state of the executing transaction, including normal execution, commit validation, commit execution and abort.

The second aspects is the extension to the directory and cache system. Two bits should be added to each cache block: the transactional read bit and transactional write bit. The transactional read bit tags whether the cache block is transactionally read by a processor, and the trasactional write bit tags whether the cache block is transactioanl written. The directory should maintain two extra bit vector for each cache block: the transactional reader bit vector and transactional writer bit vector. The transactional reader bit vector records all the processors that have performed a read operation on the cache block. The transactional writer bit vector records all the processors that have performed a write operation on the cache block.

TABLE I. SIMULATOR CONFIGRATION

Number of Cores	1 32, Sparcv9 instruction set
L1 cache	32KB, four set-associative
L2 cache	4MB
Network	2-D Mesh, 8 cycles each hop

IV. EVALUATION

A. Simulation Enviroment

We modifies the GEMS [5] simulator, a full-system simulator based on simics [6], to model a multi-core system which supports conflict graph based transactional memory. The multi-core system connects up to thirty-two processor cores through a two-dimensional mesh network. Each processor core owns its private L1 cache and part of the L2

cache. Detailed configuration of the simulator is shown in Tab. I.

We also setup two other configurations to evaluate the efficiency of CGHTM. The first one is the ideal hardware transactional memory, which is CGHTM with a network latency of zero. The second one is configured to be as close as possible to EasyHTM [7], which is also based on directory protocol.

The benchmarks used in this paper are six kernels from STAMP [8], which is a test suit developed specially for transactional memory. The six kernels are bayes, kmeans, intruder, labyrinth, ssca2 and yada. These kernels issues transactions with different length and dataset. They can reflect the performance of transactional memory system under different situation.

B. Results

Fig. 6 shows the speedup achieved by the three different transactional memory systems. In most cases CGHTM performs closer to the ideal transactional memory system.

CGHTM does not show its advantage much for two special cases. The first case is that there are a lot of circle conflicts in the benchmark, the performance of parallel processing cannot be developed. Labyrinth is an example of this case. There is a global shared array in labyrinth. Almost all the transactions read the whole array and update a few of its elements. In this case, almost arbitrary two transactions circularly conflict with each other, which introduces massive aborts. The second case is that there are few conflict among the transactions executed. In this case, all the three systems achieve near-linear speedups. Bayes and vacation are examples of this case.

V. RELATED WORK

The concept of transactional memory is introduced by herlihy [1] in 1993. After that lots of researchers propose different implementations of transactional memory, Including software implementations and hardware aided implementations. Hardware aided transactional memory supports TM semantics with special hardware resources, which serves better performance tan software transactional memory. Most of currently proposed hardware transactional memory systems, such as TCC [9], ScalableTCC [10], LogTM/LogTM-SE [11], [12], TokenTM [13], do not allow conflicting transactions both to commit, even if the serializability condition is not vaolated. This limits the performance of hardware transactional systems. Aydonat [14] proposes a hardware transactional memory implementation which supports conflict serializability through serializability order number. But it needs a huge global table to maintain metadata for all the memory blocks. This paper proposes conflict graph based hardware transactional memory, which supports conflict serializability through maintaining the conflict graph dynamically. CGHTM can achieve better concurrency with reasonable hardware cost.

VI. CONCLUSION

Conflict graph based hardware transactional memory allows two conflict transactions both to commit if serializability is not violated. This adds to the concurrency of the execution of transactions, which leads to better performance. Simulation shows CGHTM outperforms the state-of-art hardware transactional memory system for most benmarks and its performance is close to the ideal transactional memory system.

ACKNOWLEDGMENT

The author would like to thank Xuejun Yang and the anonymous reviewers, for helpful comments on this work. This work was funded by NSF grant 60921062 and 60873014.

REFERENCES

[1] M. Herlihy and J. E. B. Moss, ”Transactional memory: architectural support for lock-free data structures,” in ISCA ’93, New York, NY, USA, 1993, pp. 289–300.

[2] A. McDonald, B. D. Carlstrom, J. Chung, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun, ”Transactional Memory: The Hardware-Software Interface,” IEEE Micro, vol. 27, pp. 67-76, 2007.

[3] J. Gray and A. C. C. H. Reuter, Transaction Processing: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems), 1st ed.: Morgan Kaufmann, 1992.

[4] U. Aydonat and T. Abdelrahman, ”Serializability of Transactions in Software Transactional Memory,” 3rd Workshop on Transactional Computing, 2008.

[5] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, ”Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset,” SIGARCH Comput. Archit. News, vol. 33, pp. 92–99, 2005.

[6] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. H Aa Llberg, J. H O Gberg, F. Larsson, A. Moestedt, and B. Werner, ”Simics: A Full System Simulation Platform,” Computer, vol. 35, pp. 50–58, 2002.

[7] S. V. S. A. Tomi C, C. Perfumo, C. Kulkarni, A. A. Armejach, A. A. N. Cristal, O. Unsal, T. Harris, and M. Valero, ”EazyHTM: eager-lazy hardware transactional memory,” in MICRO 42, New York, NY, USA, 2009, pp. 145–155.

[8] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, ”STAMP: Stanford Transactional Applications for Multi-Processing,” , 2008.

[9] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, ”Transactional Memory Coherence and Consistency,” , Washington, DC, USA, 2004, p. 102.

[10] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, ”A Scalable, Non-blocking Approach to Transactional Memory,” , Washington, DC, USA, 2007, pp. 97–108.

[11] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, ”LogTM: Log-based Transactional Memory,” in Proceedings of the 12th International Symposium on High-Performance Computer Architecture, 2006, pp. 254–265.

[12] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, ”LogTM-SE: Decoupling Hardware Transactional Memory from Caches,” , Washington, DC, USA, 2007, pp. 261–272.

[13] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood, ”TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory,” , Washington, DC, USA, 2008, pp. 127–138.

[14] U. Aydonat and T. Abdelrahman, ”Hardware Support For Serializable Transactions:A Study of Feasibility and Performance,” 4th Workshop on Transactional Computing, 2009.