A Review of Software Transactional Memory in Multicore Processors

Chen Fu, Zhibo Wu, Xiaojun Wang and Xiaozong Yang
P.O. Box 1209, No.13 Fa Yuan Street, Harbin Institute of Technology, 150001, China

Abstract: The transactional memory in multicore processors has been a very hot research area over past several years. Many transactional memory systems have been proposed to solve the synchronization problem of multicore processors. Software transactional memory is one of the critical methods to ease parallel programming and improve the scalability in the environment with many cores. In this study, we give a review of the current software transactional memory systems for multicore processors. Software transactional memory systems are classified into the following categories: transaction granularity, data organization, version management, conflict detection and synchronization. Finally, we discuss an active research challenge: whether strong isolation should be supported for the tradeoff between performance and semantics correctness in software transactional memory systems.

Key words: Multicore processor, transactional memory, software, 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 developing parallel software.

Transactional memory is used to replace critical sections protected by locks in a multi-threaded parallel program. Compared with 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 (Yang et al., 2006). Third, transactions can be composed naturally, which 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, its checkpoints registers are used 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. Hardware Transactional Memory (HTM) systems (Ceze et al., 2006; Hamood et al., 2004; Herlihy and Moss, 1993; Wang et al., 2009; Yen et al., 2007) have high performance and strong atomicity. But for most HTMs, poor flexibility and additional hardware overhead are two serious disadvantages. Compared with HTM, software Transactional Memory (STM) naturally has the advantages of flexibility and easy implementation.

TRANSACTION GRANULARITY

Transaction granularity is the data store unit, through which a TM system detects conflicts. In general, STM systems have three granularity choices in reality implementation: word (or block), object and hybrid. Table 1 shows the STM systems in different transaction granularities.

Corresponding Author: Chen Fu, P.O. Box 1209, No.13 Fa Yuan Street, Harbin Institute of Technology, 150001, China
Tel: +86-451-86413354, +86-13804567649
Table 1: The STM systems in different transaction granularities

<table>
<thead>
<tr>
<th>Granularity</th>
<th>STM systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word (block)</td>
<td>Shavit STM, Haskell STM, WSTM</td>
</tr>
<tr>
<td>Object</td>
<td>DSTM, OSTM, Ananian STM, ASTM, RSTM, SXM, DSTM2, Bartok STM</td>
</tr>
<tr>
<td>Hybrid</td>
<td>TL, AUTOLOCKER, McRT-STM</td>
</tr>
</tbody>
</table>

The first choice is word (or block). Shavit and Toutou first used word (Shavit and Toutou, 1995). In their STM system, a shared word is possessed by no more than one transaction at any time. In order to guarantee a shared memory word to update atomically, a dedicate record is used to store the exclusive ownership of this word. The ownership record refers to the owner transaction of this word (Yang et al., 2005). The acquiring and releasing operations for ownership of records are performed by the system automatically. In Haskell STM, Harris etc., used block granularity (Harris et al., 2005; Perfumo et al., 2007; 2008). They use a multiword structure named TVars to store transactional variables, which include a pointer to shared data, a mutual-exclusion lock number and a wait queue used for conditional synchronization inside transactions. In WSTM Harris and Fraser (2003) map shared memory addresses into a hash table, each item of which stores a ownership record for tracking whether transactions conflict. If a transaction refers to an ownership record in the hash table, it logically refers to all the shared memory addresses hashing into that record.

The second choice is object granularity, which is used in Ananian STM, OSTM, SXM, Bartok STM, DSTM, ASTM, RSTM and DSTM2 (Ananian and Rinard, 2005; Fraser, 2004; Guerraoui et al., 2005; Harris et al., 2006; Herlihy et al., 2006; Herlihy et al., 2003a; Marathe et al., 2005, 2006). Currently, object-oriented languages are widely used in real systems. With object granularity, it is unnecessary to change original object structure for translating non-transactional program to transactional program. In addition, an object can execute inside and outside transactions without any change.

The DSTM (Herlihy et al., 2003b) is the one of the earliest object granularity STM systems. It uses a per-object runtime structure named locator to track accesses to an object. The locator has three pointers: one pointer points to a descriptor of latest active transaction, the two other pointer point to old and new versions of the data object separately. Unlike word granularity with ownership records, the locator does not hold a version number but a pointer to old and new versions of the data object. Therefore, DSTM does not need pre-knowledge of all the shared memory words accessed by a transaction to ensure ordered access, which makes DSTM itself a dynamic STM system. OSTM is another object granularity STM system (Fraser, 2004). Similar, to DSTM, OSTM uses a structure named object handle to track objects. An object handle stores a pointer to an object header, the new data object pointed by the object header and a shadow copy of the new data object, which belongs to its own transaction. In contrast to DSTM, OSTM allows multiple transactions to access the same data object and have their own shadow copy of that object. Therefore, OSTM may acquire more concurrency than DSTM when multi-transactions access the same data object.

The third choice is hybrid, which is used in TL, AUTOLOCKER, McRT-STM (Dice and Shavit, 2006; McCluskey et al., 2006; Saha et al., 2006). In these systems, transaction granularity may change between word (and block) and object: word is used when the workload has more high-level concurrent data structures (e.g., multi-dimensional arrays); object is used when the workload has more dynamical data structures.

When a STM system detects conflicts at word granularity, it supports fine-grained sharing and fine-grained parallelism. Compared with others granularity, word STM can get more concurrently access to data structures such as array, matrix etc. At the same time, it provides higher conflict detection accuracy. However, it leads to much more additional communication and associated metadata costs. These costs can cut down by increasing conflict detection granularity to block (multiword) and hash table, although they bring false conflict when no real conflicts happen, injuring performance by making unnecessary transaction aborts. Similar to block granularity, hash table not only reduce communication and ownership costs but also add false conflicts.

For programmers, object transactions are more helpful for supporting practical and dynamic object-based structures. Another benefit is that object granularity makes codes more understandable. However, it is hard to support object transactions for non-object languages which structure layouts are fixed and difficult to be changed to fit object. In addition, for high parallel data structures such as arrays, which can be logically divided into words granularity, using object for conflict detection can cause unnecessary conflicts, therefore inhibit concurrency.

Consequently, hybrid granularity STM can select the most suitable granularity according to real workload characters in order to get the best performance and it has a better trade-off for STM systems with various workloads.

**DATA ORGANIZATION**

In STM system, the methods to organize transactional data in memory can be classified into two
categories. The first category, which is used in OSTM, WSTM, DSTM, ASTM (Fraser, 2004; Harris and Fraser, 2003; Herlihy et al., 2003a; Marathe et al., 2005) organizes transactional data and ordinary data in separate memory structures. The second category, which is used in Bartok STM, RSTM, DSTM2, McRT-STM (Harris et al., 2006; Herlihy et al., 2006; Marathe et al., 2006; Saha et al., 2006) leaves transactional data in ordinary memory structure while introduce a separate structure to store metadata which are used for transactions’ concurrency control. Table 2 illustrates the STM systems in different data organizations.

To make transactional data semantics clear and keep transactional data in their own memory structure, DSTM (Herlihy et al., 2003b), OSTM (Fraser, 2004) and ASTM (Marathe et al., 2005) separate transactional data with ordinary data. Since, transactional and ordinary data are stored in different memory structure, these systems cannot access transactional data directly. If a transaction wants to access a concurrent object, it must take actions to open a TM object first. The open operations are different according to whether the access mode is READ or WRITE. If the access mode is READ, the same object body can be shared by multiple transactions at the same time. On the other hand, if the access mode is WRITE, a new version copy of the object is prepared for update and the new version copy of the object is only visible to the transaction until the transaction commits. Therefore, these systems are generally two levels of maps between the TM object and the object body: One level of map is from the TM object to an intermediate object which stores such as old and new object version references of the object body to support version management and atomic commit; Another level of map is from the intermediate object to the object body to identify the transactional data. Unfortunately each map potentially incurring a serial cache miss and seriously damage the performance of the systems (Yang et al., 2007). For this reason, ASTM reduces the levels of maps to one in READ mode rather than two as in DSTM and OSTM. However, in WRITE mode, it still needs two levels of maps.

In contrast, transactional and ordinary data are stored in the same low-level memory structure in the STM systems of the second category. They refer transactional data by ordinary pointer directly thus reduce a map via object pointer. In addition, they are convenient for spatial access locality and hence improve performance and transaction throughput.

**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). Table 3 shows the STM systems in different version managements.

In STM systems with LVM, such as TL, OSTM, WSTM, DSTM, ASTM, RSTM, Haskell STM (Diez and Shavit, 2006; Fraser, 2004; Harris and Fraser, 2003; Harris et al., 2005; Herlihy et al., 2003a; Marathe et al., 2006; Perfumo et al., 2008), old version is renamed 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 same 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.

<table>
<thead>
<tr>
<th>Table 2: The STM systems in different data organizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data organization</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Transactional and ordinary data in separate memory structures</td>
</tr>
<tr>
<td>Transactional data in ordinary memory with separate metadata</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3: The STM systems in different version managements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version management</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Lazy</td>
</tr>
<tr>
<td>Eager</td>
</tr>
</tbody>
</table>
CONFLICT DETECTION AND SYNCHRONIZATION

Besides Version Management, a STM must have the mechanisms of Conflict Detection and Synchronization to detect and resolve conflicts between concurrent transactions. 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: Eager Conflict Detection (ECD), Lazy Conflict Detection (LCD) and Hybrid Conflict Detection (HCD). Table 4 shows the STM systems in different conflict detections.

The ECD which is used in HICKS, AUTOLOCKER and Shavit STM (Hieks 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. LCD 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., 2005; Harris and Fraser, 2003; Herlihy et al., 2003; Marathe et al., 2005, 2006; Perfiano 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.

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). Table 5 shows the STM systems in different synchronizations.

Table 4: The STM systems in different conflict detections
<table>
<thead>
<tr>
<th>Conflict detection</th>
<th>STM systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eager</td>
<td>HICKS, AUTOLOCKER, Shavit STM</td>
</tr>
<tr>
<td>Lazy</td>
<td>TL, OSTM, SXM, WSTM, DSTM, ASTM, RSTM, Haskell STM</td>
</tr>
<tr>
<td>Hybrid</td>
<td>McRT-STM, Bartok STM</td>
</tr>
</tbody>
</table>

Table 5: The STM systems in different synchronizations
<table>
<thead>
<tr>
<th>Synchronization</th>
<th>STM systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonblocking</td>
<td>OSTM, SXM, WSTM, DSTM, ASTM, RSTM, Ananian STM</td>
</tr>
<tr>
<td>Blocking</td>
<td>Enmals STM, Bartok STM, AUTOLOCKER, Haskell STM, McRT-STM, TL, Haskell STM</td>
</tr>
</tbody>
</table>

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., 2005; Harris and Fraser, 2003; Herlihy et al., 2003a; 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.

The 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., 2003b). 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, Emnals STM, Bartok STM, AUTOLOCKER, Haskell STM and McRT-STM (Die and Shavit, 2006; Emnals, 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

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

Software Transactional memory provides a flexible and easy mechanism for parallel programming in multicore processors. This study presented four categories for Software transactional memory systems: transaction granularity, data organization, version management, conflict detection and synchronization. From the analysis above, we get a research trend that the future STM systems will be more likely combined with hybrid granularity, eager version management, eager or hybrid conflict detection and blocking synchronization as well as supporting strong atomicity.

ACKNOWLEDGMENTS

The authors would like to thank the guest editors and the anonymous reviewers for their detailed reviews and many constructive suggestions. This study is supported by the National High Technology Development 863 Program of China under grant No. 2006AA04A103. In the meanwhile, this work is also supported by the National Science Foundation of China under grant No. 60503015 and the Shandong Province Science and Technology Development Foundation under grant No. 2007GG10001020.

REFERENCES


