Published in final edited form as:

J Supercomput. 2016 November ; 72(11): 4379–4398. doi:10.1007/s11227-016-1747-2.

High-throughput state-machine replication using software transactional memory

Wenbing Zhao¹, William Yang², Honglei Zhang³, Jack Yang⁴, Xiong Luo⁵, Yueqin Zhu⁶, Mary Yang⁷, and Chaomin Luo⁸

¹Department of Electrical Engineering and Computer Science, Cleveland State University, Cleveland, OH 44115, USA

²Texas Advanced Computing Center, University of Texas at Austin, 10100 Burnet Road, Austin, TX 78758-4497, USA

³Agilysys Inc., Bellevue, WA, USA

⁴Division of Biostatistics and Biomathematics, Massachusetts General Hospital and Harvard Medical School, Boston, MA 02114, USA

⁵School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China

⁶Development Research Center of China Geological Survey, Key Laboratory of Geological Information Technology, Ministry of Land and Resources, Beijing 100037, China

⁷Department of Information Science, George Washington Donaghey College of Engineering and Information Technology, Joint Bioinformatics Program of University of Arkansas at Little Rock and University of Arkansas for Medical Sciences, 2801 S. University Avenue, Little Rock, AR 72204, USA

⁸Department of Electrical and Computer Engineering, University of Detroit Mercy, Detroit, MI 48221, USA

Abstract

State-machine replication is a common way of constructing general purpose fault tolerance systems. To ensure replica consistency, requests must be executed sequentially according to some total order at all non-faulty replicas. Unfortunately, this could severely limit the system throughput. This issue has been partially addressed by identifying non-conflicting requests based on application semantics and executing these requests concurrently. However, identifying and tracking non-conflicting requests require intimate knowledge of application design and implementation, and a custom fault tolerance solution developed for one application cannot be easily adopted by other applications. Software transactional memory offers a new way of constructing concurrent programs. In this article, we present the mechanisms needed to retrofit existing concurrency control algorithms designed for software transactional memory for state-machine replication. The main benefit for using software transactional memory in state-machine replication is that general purpose concurrency control mechanisms can be designed without deep

knowledge of application semantics. As such, new fault tolerance systems based on state-machine replications with excellent throughput can be easily designed and maintained. In this article, we introduce three different concurrency control mechanisms for state-machine replication using software transactional memory, namely, ordered strong strict two-phase locking, conventional timestamp-based multiversion concurrency control, and speculative timestamp-based multiversion concurrency control. Our experiments show that speculative timestamp-based multiversion concurrency control mechanism has the best performance in all types of workload, the conventional timestamp-based multiversion concurrency control offers the worst performance due to high abort rate in the presence of even moderate contention between transactions. The ordered strong strict two-phase locking mechanism offers the simplest solution with excellent performance in low contention workload, and fairly good performance in high contention workload.

Keywords

State-machine replication; Software transactional memory; Ordered strong strict two-phase locking; Multiversion concurrency control; One-copy serializability

1 Introduction

State-machine replication is a common way of constructing general purpose fault tolerance systems. To ensure replica consistency, requests must be executed sequentially according to some total order at all non-faulty replicas [24]. Unfortunately, this could severely limit the system throughput. In the past several years, we have seen great efforts on the partial removal of this constraint. Most of them rely on using application semantics to develop custom replication algorithms that could identify independent requests and execute them in parallel [26]. Previously, we demonstrated that using software transactional memory, it is possible to enable concurrent execution of requests automatically if they contain no conflicting operations [20]. This is an attractive approach to increasing the system throughput because no application-specific rules are required to determine whether or not two requests are conflicting. Even if two concurrently executing requests actually contain conflicting operations, both might be able to commit if the relative ordering of all conflicting operations is identical and is consistent with the commit order. Only in the worst cases, one of them will have to be aborted and restarted by the system runtime.

In this article, we describe concurrency control mechanisms for state-machine replication using software transactional memory in detail. We show that standard concurrency control mechanisms designed for stand-alone software transactional memory-based applications cannot be used directly to ensure one-copy serializability. To ensure replica consistency and one-copy serializability, we require that all transactions be committed in a total order. When a transaction is ready to be committed, it first blocks until all transactions that have been ordered ahead of it have been committed, and then validates its operations according to predefined validation rules. The transaction is committed if the validation is successful, and is rolled back if the validation fails. An aborted transaction is retried until it can be committed.

We investigate three different concurrency control mechanisms for state-machine replication using software transactional memory, namely, ordered strong strict two-phase locking, conventional timestamp-based multiversion concurrency control, and speculative timestamp-based multiversion concurrency control. Our experiments show that speculative timestamp-based multiversion concurrency control mechanism has the best performance in all types of workload, the conventional timestamp-based multiversion concurrency control offers the worst performance due to high abort rate in the presence of even moderate contention in the workload. The ordered strong strict two-phase locking mechanism offers the simplest solution with excellent performance in low contention workload, and fairly good performance in high contention workload.

In summary, we make the following research contributions:

- We formally introduce the system model and define the correctness criteria for using software transactional memory in state-machine replication.
- We define the concurrency control mechanism for state-machine replication based on two-phase locking.
- We define the concurrency control mechanism for state-machine replication based on multiversion concurrency control.
- We perform a micro-benchmark on the mechanisms we have proposed and discuss why some is working better than others.

2 Background and related work

Tremendous efforts have been made in the past decade to increase the performance of fault tolerance systems [4,6–9,19,25,27]. A review of this line of research is given in [26]. The primary approach to increasing the system performance is to enable concurrent execution of independent requests. Typically, application semantics is needed to develop a set of rules that can be used to dynamically determine whether or not two requests have inter-dependency. Unfortunately, it is expensive to discover appropriate application semantics and it is hard to maintain such a system for possible changes [23]. Furthermore, rules developed for one application cannot be easily applied to other applications.

Software transactional memory [17] provides an alternative opportunity to enable concurrent execution of requests dynamically without resorting to the use of predefined application-specific rules. In this approach, each request would trigger a new atomic transaction that consists of a sequence of read and write operations on a set of data items. Existing software transactional memory implementations can be roughly divided into two camps: (1) two-phase locking (2PL) based, and (2) nonblocking based.

2PL-based approach: Traditionally, a transaction must acquire a lock for each datum before the transaction accesses it, and this approach has been used for software transactional memory [11,13]. There are several flavors of 2PL. In plain 2PL, a transaction acquires locks as it needs them. However, it must not acquire any more locks as soon as it has released the first lock it has held before. Hence, in plain 2PL, there is an expanding phase in terms of the

number of locks acquired, and a shrinking phase when the number of locks decreases. In strict 2PL, a transaction must release all write locks only after it is committed or aborted while it can release a read lock once it finishes reading the datum protected by the read lock. The strongest form of 2PL is called strong strict 2PL, in which a transaction must release all its locks (read and write locks) at once at the end of the transaction.

In software transactional memory research, a speculative form of 2PL was proposed in [10]. In this scheme, a transaction may perform the write and read operations without first acquiring the corresponding locks, and it waits until the commitment time to acquire the locks for its write set, and validates its read set. If the validation is successful, the transaction is committed and the new values for the write set are stored in place in memory.

A major challenge for the 2PL-based approach is deadlock detection and recovery. A transaction may also be blocked if the locks for the write set are not available (i.e., the locks are held by another transaction).

Nonblocking-based approach: In this approach, a transaction can perform read and write on the data set without blocking, typically using multiple versions of each datum. Locks are still needed to protect shared meta-data structure for the runtime. The commit order can be determined either based on the timestamps of the transactions, or based on the dependency between conflicting transactions [16] (for non-conflicting transactions, their relative commit order is not important).

Regardless of the approaches used, a software transactional memory framework always ensures serializable execution of concurrent transactions. Hence, in the presence of read/write, write/read, and write/write conflicts between each pair of concurrent transactions, they all have to follow the same relative order, otherwise, one of them will have to be aborted [12]. To use software transactional memory for state-machine replication, we must provide additional mechanisms to ensure the one-copy serializable execution property, i.e., the set of transactions executed at all non-faulty replicas must use identical serializable execution schedule.

An intuitive idea to satisfy the one-copy serializable property in a state-machine replication system is to first totally order incoming requests, and then ensure that all transactions triggered by the requests are committed in exactly the same total order. This idea was first applied to active replication with the crash-fault model [2], and later for Byzantine fault tolerance in [20]. In both works, some existing software transactional memory frameworks were used, but no details were presented regarding what concurrency control mechanisms are used and how the total commit order is reinforced. In fact, it is not straightforward to implement the idea with such frameworks because:

- These software transactional memory frameworks may use different conflict resolution rules. Hence, the mechanisms to ensure one-copy serializable property must be customized for them.
- In these rules, the decision to commit or abort a transaction is solely based on the concern of ensuring the linearizability of transactions without regard to any

predefined ordering of these transactions. In state-machine replication, transactions that could be committed out of order must be aborted because non-faulty replicas may become inconsistent if a transaction is committed at a non-faulty replica in one order while it is committed at another replica in a different order.

- In these frameworks, if a transaction is aborted, it would be retried *as a new transaction*. Unfortunately, to ensure the total commit order of all transactions, this is not acceptable. A transaction must be committed in the total order at all non-faulty replicas even if it has to be retried.

The objective of this paper is to define the correctness requirement in using software transactional memory for concurrent state-machine replication, and present the mechanisms that could be used to satisfy the requirement in detail.

3 System model

We assume that the system operates in an asynchronous distributed environment. The server is replicated for fault tolerance. A client sends a request to the replicated server and then synchronously waits for the corresponding reply. To ensure replica consistency, the requests must be totally ordered. Any existing distributed agreement algorithm or replication algorithm, such as Paxos [14,30] (for crash fault tolerance), and PBFT [3,22] (for Byzantine fault tolerance), could be used to totally order these requests.

We advocate to separate agreement from execution [18] where the total ordering of requests is accomplished using a separate agreement cluster, and the replicas are dedicated to request processing only. The number of agreement nodes needed depends on the algorithms used for total ordering. Typically, $2f + 1$ nodes are needed to achieve crash fault tolerance, and $3f + 1$ nodes are needed for Byzantine fault tolerance, where f is the maximum number of faulty nodes tolerated. Furthermore, for Byzantine fault tolerance, each message is additionally protected by a security token such as digital signature or message authentication code to ensure the integrity of the messages exchanged. Due to the separation of agreement and execution, fewer number of replicas are needed. To tolerate crash faults, $g + 1$ replicas are needed to tolerate up to g faulty replicas. To tolerate Byzantine faults, $2g + 1$ replicas are needed to tolerate up to g faulty replicas. Each replica would need to receive at least $f + 1$ matching ordered messages from different agreement nodes to accept the binding of the total order and each request for crash fault tolerance.

We assume that the server application is constructed using software transactional memory. For each request from a client, one and only one atomic transaction is started. A transaction may be aborted if a conflict is detected. However, an aborted transaction will be retried automatically as soon as it is aborted until it is finally committed.

For correct operation, the replicated server must ensure one-copy serializability. Specifically, the following safety property must be guaranteed:

Safety property: Transactions executed at all non-faulty replicas must commit in the same total order and the order is equivalent to the order when these transactions are executed sequentially at a single process.

Let m_i be the request message that has a sequence number i . The sequence number i indicates the total order for the message, i.e., m_i is *supposed* to be executed at all non-faulty replicas after all previously ordered requests, $m_0, m_1, \dots, m_{i-1}$, have been executed. For each m_i , a transaction T_i is started, where i is assigned to the transaction as its identifier and timestamp.

A transaction has one or more read and write operations on a set of data items. Let $R(d)_i$ be the read operation on datum d issued by transaction T_i . Similarly, let $W(d)_i$ be the write operation on datum d issued by transaction T_i . Two transactions T_i and T_j are conflicting with each other if any of the following is true:

- Read/write conflict: T_i (or T_j) issues a read operation $R(d)_i$ (or $R(d)_j$) and T_j (or T_i) subsequently issues a write operation $W(d)_j$ (or $W(d)_i$).
- Write/read conflict: T_i (or T_j) issues a write operation $W(d)_i$ (or $W(d)_j$) and T_j (or T_i) subsequently issues a read operation $R(d)_j$ (or $R(d)_i$).
- Write/write conflict: T_i issues a write operation $W(d)_i$ and T_j concurrently issues another write operation $W(d)_j$ (it doesn't matter which transaction issues the write operation on the same datum first).

If we consider the linearizability requirement only, conflicting transactions could all be committed if all conflicting operations happen to be executed in the same order. Otherwise, one of them will have to be aborted. For state-machine replication, however, this requirement is inadequate. To ensure one-copy serializability in a fault tolerance system, transactions that might be able to commit may have to be aborted if the linearized order of two conflicting transactions is inconsistent with the total order.

4 Concurrency control mechanisms for state-machine replication

In the context of state-machine replication, one-copy serializability can be achieved by first establishing a total order of all requests, and subsequently committing transactions triggered by these requests according to this total order. More specifically, given a transaction T_i , it can be committed if and only if all transactions T_j where $j < i$ have been committed. Before a transaction can be committed, it must be validated. The validation rules heavily depend on the concurrency control method used. The transaction is aborted and retried if the validation fails. In the following subsections, we discuss mechanisms for the two popular approaches of implementing software transactional memory as we have outlined in Sect. 2.

4.1 Mechanisms for 2PL-based concurrency control

In this section, we discuss mechanisms needed for plain 2PL, strict 2PL, and strong strict 2PL to ensure one-copy serializability of transactions. We do not consider the speculative

2PL-based approach introduced in [10]. Extending it for one-copy serializability is left for future work.

In the flavors of 2PL that we consider, a read operation $R(d)_i$ by T_i would read the last committed value of d , and a write operation $W(d)_i$ by T_i will not be visible to other transactions until T_i has been committed. During the validation step, the following validation rule is used:

Validation rule: A transaction T_j is aborted and retried if T_j issued a read on some datum d (i.e., $R(d)_j$), and there exists a write on the same datum d (i.e., $W(d)_i$) issued by another transaction T_i , where T_i is ordered after the last committed transaction T_k that wrote to d at the time $R(d)_j$ is issued (i.e., $k < i < j$).

This is because d has already been modified by some older, but uncommitted, transaction, and hence, the value returned by $R(d)_j$ would be an obsolete value. Hence, T_j must be aborted and then retried. The retry would be successful if T_j has been committed by the time T_j issues the $R(d)_j$ operation. An example is shown in Fig. 1. We assume that the value of d is d_k prior to the $R(d)_j$ operation is issued. In T_i , a write operation is issued on d , which changes d to d_i . In T_j , a read operation on d is issued, which would return d_k . T_j will not see the uncommitted value for d written to by T_i . We have to abort T_j because not doing so would violate the causality of the system (note that T_j is required to commit after T_i according to the total order). After T_j is retried, the most recent value d_i , which is written to by T_i , is returned to the read operation, thereby, preserving the causality of the system, and the retried transaction T_j can now be committed.

Note that from the linearizability point of view, T_j does not need to be aborted if it is read-only. However, in the context of state-machine replication, this is not allowed because we must ensure the causality of the system, i.e., if T_{i-1} has updated a datum d , and T_j reads d , then the updated value of d from T_{i-1} must be read by T_j .

Also note that if T_j has issued a write operation on d that some older transaction T_i ($i < j$) has read, then T_j can still be committed because T_i will not read from the modified value of d by T_j .

Similarly, if T_i and some younger transaction T_j ($i < j$) have all written to a datum d , both T_i and T_j can be committed because of the total commit ordering. The value written by T_i will always be overwritten by that of T_j , regardless of which transaction has issued the write operation first. This policy would not violate any causality constraint because a read operation would have to be involved for any information to be propagated from one transaction to another, which our validation rule already addresses. In some implementation, a write is always interpreted as both a read and a write. In this case, our validation rule on the read-write/write-read conflict is sufficient.

Finally, the validation step is necessary only if plain 2PL and strict 2PL are used. If strong strict 2PL is used, it is impossible for two uncommitted concurrent transactions to hold the

locks for the same datum (i.e., one cannot acquire a read lock while the other transaction is holding the write lock for the same datum).

In software transactional memory implementations, there is no restriction in place as to when a transaction is allowed to acquire (or to attempt to acquire) locks. Typically, either a transaction acquires locks as it progresses, or acquires all the locks at the beginning of a transaction. Most importantly, different transactions are allowed to compete for locks freely. This could create unpredictable interleavings in lock acquisition, which is prone to deadlocks. Our preliminary experiment result confirms this hypothesis. We have observed frequent deadlocks even if transactions access very few shared data items (10 or less). Hence, we propose that all transactions acquire all necessary locks according to the total order, i.e., the same order for requests delivery and transaction commitment, and release them only when the transaction is committed or aborted. This ordered strong strict 2PL variation ensures deadlock-free and has excellent performance, as we will show in Sect. 5.1. Another benefit of using ordered strong strict 2PL is that no validation is necessary (i.e., all transactions are guaranteed to be committed).

4.2 Mechanisms for nonblocking concurrency control

Most nonblocking concurrency control mechanisms in software transactional memory are derived from the multiversion concurrency control [1]. A major advantage of multiversion concurrency control is that a read-only transaction can be committed successful without blocking. A transaction is never forced to rollback when it issues a read operation. This is accomplished by maintaining multiple versions of a datum. When a write is successful, a new version of the datum is created. The availability of multiple versions of each datum makes it possible to find an appropriate version for each read operation to ensure the serializability of the read transaction with respect to other transactions.

A common implementation of multiversion concurrency control is to apply a timestamp ordering to transactions. In this approach, each transaction is assigned a monotonically increasing timestamp and transactions must commit in the order of their timestamps. This design aligns very well with the one-copy serializability requirement for state-machine replication *if all transactions can be committed*. However, it is inevitable that some transactions are aborted. On retry of aborted transactions, they are treated as new transactions with new timestamps. Unfortunately, this could cause replica inconsistency because depending on when the conflicting operation is issued, a transaction that issues a write operation might be able to commit at one replica, while the same transaction executed at another replica might have to be rolled back. Unlike what is defined in multiversion concurrency control, when retrying a transaction, we cannot alter the order of its commitment with respect to other transactions to preserve the replica consistency.

As shown in Fig. 2, according to multiversion concurrency control, each datum maintains a list of versions, and for each version, there are two associated timestamps, RTS, and WTS. RTS is the largest timestamp of the transaction that has read the datum. WTS is the timestamp of the transaction that created the version due to a write to the datum. For example, if transactions T_i and T_j both read version 2 of datum x and $j > i$, then $\text{RTS}(x_2) = j$, and if transaction T_k is the transaction that has written to x for version 2, then $\text{WTS}(x_2) = k$.

Here we assume that the subscript of a transaction denotes its assigned timestamp, i.e., $TS(T_i) = i$.

4.2.1 Conventional multiversion concurrency control—In conventional multiversion concurrency control, a read operation on datum d issued by a transaction T_k would return the version i of d that was created by the most recent committed version prior to the read operation, or created within the current transaction. Specifically, the following defines the read rule and the write rule.

Read rule: Read of d by T_k would return the version that it has written to within the current transaction. If T_k has not written to d , the read returns the version vi of d , where $WTS(d_{vi}) = s$, and s is the timestamp of the most recent committed transaction. Subsequently, $RTS(d_{vi})$ is set to be the maximum of the current read timestamp of d_{vi} and the timestamp of the current transaction, i.e., $RTS(d_{vi}) = \max(RTS(d_{vi}), k)$.

Write rule: For a write on d issued by transaction T_k , a new version of d is created with a version number $vi = \max(vj) + 1$, for all existing version vj of d (both committed and uncommitted versions). The read timestamp and the write timestamp of the new version is set to the timestamp of the transaction, i.e., $RTS(d_{vi}) = WTS(d_{vi}) = k$.

According to traditional timestamp-based multiversion concurrency control, a transaction T_k must be aborted if it issues a write on a datum d and the following condition is met:

- Among all versions of d where $WTS(d) \leq TS(T)$, there exists at least one version vi of d where $RTS(d_{vi}) > TS(T)$.

This means that a version of d that could be read by T_k legally has been read by a younger transaction T_s than T_k , i.e., $s > k$. The reason why T_k has to be aborted is that allowing the write operation on d would create a dependency from T_s to T_k . This could make T_k and T_s not serializable, if T_k has accessed some shared data items before T_s . The benefits of this mechanism is that a transaction could be aborted at the earliest possible moment on a write operation that could impact the serializability of concurrent transactions. However, this mechanism does not work for state-machine replication. This is because all transactions must be committed according to a total order at all non-faulty replicas. This mechanism favors younger transactions, i.e., the transactions that are started later, by aborting older transactions. Due to the total commit order, younger transactions are not allowed to commit while an older transaction is aborted and retried. Unfortunately, if the younger transaction T_s is not aborted, the older transaction T_k cannot be successfully retried because it assumes its original timestamp in state-machine replication instead of getting a new timestamp during retries. Hence, we must abort and retry the younger transaction if it has read a shared datum that is written to by an older transaction (either before or after the read operation), that is, T_s would be aborted and retried and T_k would be allowed to commit.

The validation rule enforced at the commitment time is, therefore, given below:

Validation rule: A Transaction T_s is aborted if it has issued a read on any committed version v_j of datum d , and another uncommitted transaction T_k , where $k < s$, has written to d , i.e., there exists a version v_i of d where $\text{RTS}(d_{v_j}) > \text{WTS}(d_{v_i})$ and $v_i \neq v_j$.

Note that conventional timestamp-based multiversion concurrency control does not allow the read of uncommitted values. That is why the version v_j of d read by T_s is guaranteed to be different from (i.e., smaller than) the version v_i that is written to by T_k . Compared with the original abort rule in multiversion concurrency control, we cannot abort a transaction until the commitment time, which could limit the throughput of a state-machine replicated system. This is an inevitable price to pay for fault tolerance.

4.2.2 Speculative multiversion concurrency control—We could reduce the abort rate if we allow the read of uncommitted data written to by a different transaction. To see why this is the case, consider our validation rule for the conventional multiversion concurrency control. According to our validation rule, the current transaction would be aborted if it has read a version of the datum and an uncommitted older transaction has concurrently written to the same datum. When the validation fails for a transaction, there are actual two types of conflicts, as illustrated in Fig. 3:

- Read–write conflict: the current transaction reads the datum first, then uncommitted older transaction did a write to the same datum.
- Write–read conflict: an uncommitted older transaction did a write to the datum, then the current transaction did a read on the same datum. According to the read rule of the conventional multiversion concurrency control, the uncommitted version of d is not visible to the current transaction. Hence, the most recent committed version is returned to the current transaction.

As we can easily see, if the second scenario (i.e., write–read conflict) happens, we could have made the uncommitted version written to the datum by an older transaction visible to the current transaction. In this way, the schedule of the two conflicting transactions would be serializable and their relative ordering would be consistent with our total ordering requirement. Hence, the current transaction could be allowed to commit in the presence of write–read conflicts if the uncommitted written version is made visible to the read.

We should note that making an uncommitted version visible to a younger transaction is not without its risk. If the uncommitted transaction that produced the version is eventually aborted, then all transactions that have read the version must also be aborted. That is why we refer to this variation of multiversion concurrency control as speculative multiversion concurrency control.

In speculative multiversion concurrency control, the read rule is changed to allow the read of uncommitted value that is issued by an older transaction:

Read rule: A read operation on datum d issued by a transaction T_k would return the version v_i of d that was created by the most recent version (regardless of committed or uncommitted) prior to, or within, the current transaction.

The validation rule remains the same as that for conventional multiversion concurrency control.

5 Implementation and performance evaluation

The proposed concurrency control mechanisms for state-machine replication are implemented in C++. Standard pthread application programming interfaces are used for locking (using regular, write and read locks) and for enforcing the total order of commitment (using conditional variables). One thread is designated to handle incoming requests, and a pool of worker threads is used to process the transactions triggered by the incoming requests, as illustrated in Fig. 4. Each datum is represented by a C++ object encapsulating the datum value, locks, and all necessary meta-data such as version information. For multiversion concurrency control, a linked list is used to keep track of different versions (one node per version).

Each transaction maintains a read set and a write set. For multiversion concurrency control, a transaction also keeps track of the versions of data that it has read from (including each version's RTS and WTS timestamps) for the purpose of validation.

For performance evaluation, we have experimented with different number of threads with different number of shared data items. In each run, every thread continuously executes 1000 transactions, and the throughput is averaged across these transactions. Although the number of shared data items is predetermined, the order in which a thread accesses these data items is randomly chosen so that typically different threads access the same shared datum in different order. For each transaction, we assume that half of the shared data items will be written to, and half are to be read from. We intentionally choose the 50–50% split on read and write because it gives the maximum possibility of conflict between concurrent transactions. Hence, the results presented in this section represent the worst-case scenarios. For each transaction, we inject 5m processing time to simulate actual load.

The performance of our concurrency control mechanisms is assessed by the speedup ratios (or speedup for short) under various number of shared data items between concurrent transactions. The speedup is defined to be the ratio of the throughput obtained in a multithreaded system and that of a single-threaded system. A perfectly functioning concurrency control mechanism would lead to a speedup linearly proportional to the number of threads (i.e., the speedup should be n if n threads are used to concurrently execute transactions) in the absence of conflicts. In the presence of conflicts between concurrent transactions, no solution could attain perfect speedup. Specifically, a traditional implementation of state-machine replication would execute all request sequentially, which means that there is no concurrent processing regardless the number of cores exist in the CPU, i.e., the speedup would be 1 for all scenarios that we experiment.

The experiments are done using an iMac computer that is equipped with a quad-core 3.4GHz i7-2600 CPU, and 12GB of RAM. With hyperthreading, the CPU offers a total of 8 logical cores. According to our design, 1 thread is designated for job allocation, which will occupy one of the logical cores. This leaves the remaining 7 logical cores available for

executing transactions concurrently. Hence, we perform experiments with up to 7 threads. In this section, we present the micro-benchmark result for the mechanisms we have introduced in Sect. 4. In addition to varying the number of threads, we vary the number of shared data between different concurrent transactions from 0 (no sharing) to 100. Due to the hardware limitation, we cannot assess the scalability of our mechanisms in terms of the number of concurrent transactions. Note that due to the lack of standard benchmark suite for state-machine replication, it is common practice in fault tolerance system research to report only micro-benchmark results [4,19,21,28,29].

5.1 Performance of ordered strong strict 2PL

The average throughput for different number of concurrent threads with different number of shared data items is shown in Fig. 5. As can be seen, the ordered strong strict 2PL works very well for small number of shared data items. The speedup for 10 shared data items is virtually identical to that when concurrent transactions are working on disjoint data items (which we refer to as the base case). When the number of shared data items increases, the speedup in throughput does reduce gradually. However, the reduction in speedup is quite resilient to the number of shared data items. Even when the number of shared data items reaches 100, the speedup reduces by less than half compared with the base case.

Nevertheless, to find out the primary factor that causes the speedup reduction, we measure the wait time for each transaction to get its turn to acquire the locks (which we refer to as LWait time), and the wait time for each transaction to get its turn to commit (which we refer to as CWait time). As can be seen in Fig. 6, the dominating factor is the LWait time. For larger number of threads and larger number of shared data items, the LWait time could go as high as 5 ms. The increase in LWait time is apparently nonlinear with respect to the number of threads and the number of shared data items. It is also interesting to note that since the threads are synchronized by the total order of acquiring locks, the CWait time is very little (under 40 μ s in all scenarios). This is not surprising because we apply the same processing time for all transactions. If the processing time varies across different transactions, we expect the CWait time to be comparable to the LWait time.

5.2 Performance of conventional multiversion concurrency control

The performance of conventional multiversion concurrency control is shown in Fig. 7. As can be seen, the speedup is incredibly low even when the number of shared data item is as low as 10. For example, the speedup is less than 2.0 when the number of threads is 7 with 10 shared data items. Ideally, the speedup should be 7.0 under this condition.

The very poor performance of the conventional multiversion concurrency control is due to high abort rate, as shown in Fig. 7. With 10 shared data items, even if there are only 2 threads, virtually all transactions would have to be aborted and retried. It is not clear to us why as the number of threads increases from 2 to 7, the abort rate actually reduces. The reduction trend is most prominent when the number of shared data items ranges between 10 to 50.

5.3 Performance of speculative multiversion concurrency control

By making uncommitted write visible to younger transactions, the abort rate can be reduced significantly (by a factor of about 10), as shown in Fig. 8. As a result, the throughput speedup is improved dramatically, and it is very resilient to the number of shared items between transactions, as shown in Fig. 8. With 100 shared data items, the speedup is 5.26, which is much higher than the speedup of 1.14 for conventional multiversion concurrency control under the same condition.

5.4 Discussion

As can be seen, the conventional multiversion concurrency control is clearly a not good fit for state-machine replication due to its very high abort rate in the presence of small number of shared items between transactions. Among the three concurrency control mechanisms that we have considered, the speculative multiversion concurrency control offers the best performance. Even if all transactions content for the same set of 100 data items, speculative multiversion concurrency control could achieve a speedup of 5.26, while the ordered strong strict 2PL could achieve only 3.72. These two concurrency control mechanisms offer similar speedup in low contention rates. It is worth noting the 5.26 speedup observed for speculative multiversion concurrency control with 100 shared data between concurrent transactions is 75% of the maximum speedup possible even if there is no conflict between concurrent transactions. Hence, we believe that this speedup is near optimal.

Our observation is consistent with studies on stand-alone software transactional memory implementation. In [15], authors employed a concurrency control mechanism similar to the speculative multiversion concurrency control and reported significantly better speedup than other approaches. In [10], authors proposed a speculative form of strong strict 2PL concurrency control mechanism, and reported good performance. We choose not to use the speculative form of strong strict 2PL in our implementation for simplicity. By imposing a total order on lock acquisitions for different transactions, we could eliminate deadlocks and the ensuing aborts of transactions. Essentially, we make a tradeoff to incur wait time for lock acquisitions rather than dealing with deadlocks. We doubt that the performance could be improved if speculation is employed for strong strict 2PL. As we have shown in our experiments, high abort rate is detrimental to the system performance.

We now take this opportunity to summarize how the challenges we raised in Sect. 2 regarding using software transactional memory in state-machine replication:

- Challenge: these software transactional memory frameworks may use different conflict resolution rules. Hence, the mechanisms to ensure one-copy serializable property must be customized for them. Our solution: we presented conflict resolution mechanisms for two major concurrency control algorithms, namely, two-phase commit, and multiversion concurrency control. All existing concurrency control algorithms are based on these two algorithms. Hence, this work present a useful guide regarding how to customize applications for state-machine replication.

- Challenge: in these rules, the decision to commit or abort a transaction is solely based on the concern of ensuring the linearizability of transactions without regard to any predefined ordering of these transactions. In state-machine replication, transactions that could be committed out of order must be aborted because non-faulty replicas may become inconsistent if a transaction is committed at a non-faulty replica in one order while it is committed at another replica in a different order. Our solution: our conflict resolution rules explicitly consider the relative ordering of conflicting transactions. More specifically, we choose to abort the younger transaction even if it could be committed if it is re-ordered ahead of an older transaction. The tradeoff is that we might abort more transactions in state-machine replication. This is an inevitable price to pay to presence replication consistency.
- Challenge: in these frameworks, if a transaction is aborted, it would be retried *as a new transaction*. Unfortunately, to ensure the total commit order of all transactions, this is not acceptable. A transaction must be committed in the total order at all non-faulty replicas even if it has to be retried. Our solution: our mechanisms ensure that when a transaction is retried, it assumes the same transaction id instead of being assigned a new one. This ensures that all transactions commit in the same total order at all replicas even some of the transactions are aborted and retried.

So far, we have not explicitly compared our approaches with competing approaches. At the low end of the spectrum, one approach is to adhere to the state-machine replication requirement, where all requests are processed sequentially. In this case, the availability of multiple cores in CPU is not utilized and the speedup would also remain to be 1 regardless how many threads are launched because they cannot run concurrently. Previously, we have studied extensively on how to enable concurrent execution of requests by exploiting application semantics. This approach can be considered “handcrafted concurrency” and we have achieved moderate success. At the other end of the spectrum is a hypothetical full degree of concurrency where the speedup would be n for n threads. In Fig. 9, we show a comparison of our approaches with respect these three alternatives. The figure includes the following five scenarios:

- The traditional finite-state-machine approach where the speedup remains at 1. It is labeled as “Traditional”.
- The result we obtained [7] where the concurrent execution is achieved via the use of conflict-free replicated data types. The result is labeled as “handcrafted”.
- The result we obtained in this research using ordered strong strict 2PL with 100 shared data items. It is labeled as “OSS2PL”.
- The result we obtained in this research using speculative multiversion concurrency control with 100 shared data items. It is labeled as “SpecMVCC”.
- A hypothetical result with full speedup. It is labeled “Ideal”.

As can be seen, the optimized versions of our approaches show superior performance compared with the “handcrafted concurrency” approach.

Another issue we have yet to discuss is failure recovery. Due to the separation of agreement from execution [5,18], the replicas do not participate in the total ordering of requests. All replicas are equal in that no replica makes decision for other replicas. Hence, the execution of requests is completely local to each replica once the total ordering of the requests has been received from the agreement cluster, and consequently, the failure recovery is local as well. Most appealingly, the failure of up to g replicas would have no direct impact on the clients of the replicated system because one or more surviving replicas would be able to provide the clients with replies, i.e., the failure and failure recovery are not on the critical path of request execution.

The local failure recovery mechanism is rather simple. On restart after a failure, a replica would abort all ongoing transactions and restart them for recovery. Hence, the local recovery time depends on the latency to commit the restarted transactions. Because of the nature of the software transactional memory approach, transaction restart is expected and no additional mechanism is required to handle failure recovery.

Finally, in this study, we presented the result using a micro-benchmark. For fault tolerance system research, this is common practice because there is no standard benchmark available to compare different solutions. Also note that the 50–50% split on read and write is in fact the worst-case scenario because this setup gives the maximum possibility of conflicts between concurrent transactions. Other read–write mix would show better performance.

6 Conclusion and future work

In this article, we presented the detailed concurrency control mechanisms for state-machine replication using software transactional memory. We first define the correctness property for such mechanisms, we then explained why the built-in concurrency control mechanisms in existing software transactional frameworks cannot be used directly to achieve one-copy serialization in state-machine replication. This is followed by the elaboration of three concurrency control mechanisms that can be used to enable concurrent execution in state-machine replication-based fault tolerance systems using software transactional memory. Our experiments show that speculative timestamp-based multiversion concurrency control mechanism has the best performance in all types of workload, the conventional timestamp-based multiversion concurrency control offers the worst performance due to high abort rate in the presence of even moderate contention. The ordered strong strict two-phase locking mechanism offers the simplest solution with excellent performance in low contention workload, and fairly good performance in high contention workload.

Finally, although we have demonstrated that software transactional memory can be used to achieve high throughput in state-machine replication, there are challenges in adopting this approach for practical systems. The primary obstacle is that an application would have to be completely rewritten using the software transactional memory model. Furthermore, existing software transactional memory libraries would have to be enhanced with the mechanisms proposed in this article. To alleviate such limitations, software tools are needed to automatically transform an existing application to one that uses software transactional

memory, and popular software transactional memory libraries should be instrumented with our mechanisms so that they are ready to be used for state-machine replication.

Acknowledgments

We sincerely thank the anonymous reviewers for their invaluable comments and suggestions. An earlier version of this paper was presented at the 8th International Conference on Advanced Software Engineering & Its Applications [31]. This work was supported in part by a Graduate Faculty Travel Award from Cleveland State University, by the National Key Technologies R&D Program of China under Grant 2015BAK38B01, and by a grant from the special funds project for scientific research of public welfare industry from the Ministry of Land and Resources of the Peoples Republic of China No. 201511079.

References

1. Bernstein PA, Goodman N. Concurrency control in distributed database systems. *ACM Comput Surv.* 1981; 13(2):185–221.
2. Brito, A., Fetzer, C., Felber, P. Multithreading-enabled active replication for event stream processing operators. *Proceedings of the 28th IEEE international symposium on reliable distributed systems*; New York: IEEE; 2009. p. 22–31.
3. Castro M, Liskov B. Practical byzantine fault tolerance and proactive recovery. *ACM Trans Comput Syst.* 2002; 20(4):398–461.
4. Chai H, Zhang H, Zhao W, Melliar-Smith PM, Moser LE. Toward trustworthy coordination for web service business activities. *IEEE Trans Serv Comput.* 2013; 6(2):276–288.
5. Chai, H., Zhao, W. Byzantine fault tolerance as a service. In: Kim, ThMohammed, S.Ramos, C.Abawajy, J.Kang, BH., Slezak, D., editors. *Computer applications for web, human computer interaction, signal and image processing, and pattern recognition, communications in computer and information science.* Vol. 342. Springer; Berlin: 2012. p. 173–179.
6. Chai H, Zhao W. Byzantine fault tolerance for session-oriented multi-tiered applications. *Int J Web Sci.* 2013; 2(1/2):113–125.
7. Chai, H., Zhao, W. Byzantine fault tolerance for services with commutative operations. *Proceedings of the IEEE international conference on services computing*; Anchorage: IEEE; 2014. p. 219–226.
8. Chai, H., Zhao, W. Byzantine fault tolerant event stream processing for autonomic computing. *Proceedings of the 12th IEEE international conference on dependable, autonomic and secure computing*; New York: IEEE; 2014. p. 109–114.
9. Chai, H., Zhao, W. Towards trustworthy complex event processing. *Proceedings of the 5th IEEE international conference on software engineering and service science*; New York: IEEE; 2014. p. 758–761.
10. Dice, D., Shalev, O., Shavit, N. *Distributed computing.* Springer; Berlin: 2006. Transactional locking. II; p. 194–208.
11. Ennals, R. Tech rep, technical report IRC-TR-06-052, Intel research cambridge tech report. 2006. Software transactional memory should not be obstruction-free.
12. Gray, J., Reuter, A. *Trans Process Conc Techniq.* 1. Morgan Kaufmann Publishers Inc; San Francisco: 1992.
13. Harris, T., Fraser, K. *ACMSIGPLAN notices.* Vol. 38. ACM; New York: 2003. Language support for lightweight transactions; p. 388–402.
14. Lamport L. Paxos made simple. *ACM SIGACT News (Distrib Comput Column).* 2001; 32(4):18–25.
15. Ramadan, HE., Roy, I., Herlihy, M., Witchel, E. *ACM sigplan notices.* Vol. 44. ACM; New York: 2009. Committing conflicting transactions in an stm; p. 163–172.
16. Ras, Y. The principle of commitment ordering. *Proceedings of the 18th international conference on very large data bases*; 1992. p. 292–312.
17. Shavit, N., Touitou, D. Software transactional memory. *Proceedings of the 14th ACM symposium on principles of distributed computing*; 1995. p. 204–213.

18. Yin, J., Martin, JP., Venkataramani, A., Alvisi, L., Dahlin, M. Separating agreement from execution for byzantine fault tolerant services. Proceedings of the ACM symposium on operating systems principles; Bolton Landing, NY. 2003. p. 253-267.
19. Zhang H, Chai H, Zhao W, Melliar-Smith PM, Moser LE. Trustworthy coordination for web service atomic transactions. IEEE Trans Parall Distrib Syst. 2012; 23(8):1551–1565.
20. Zhang H, Zhao W. Concurrent byzantine fault tolerance for software-transactional-memory based applications. Int J Future Comput Commun. 2012; 1(1):47–50.
21. Zhang H, Zhao W, Melliar-Smith PM, Moser LE. Design and implementation of a byzantine fault tolerance framework for non-deterministic applications. IET Softw. 2011; 5:342–356.
22. Zhao W. Design and implementation of a Byzantine fault tolerance framework for web services. J Syst Softw. 2009; 82(6):1004–1015.
23. Zhao, W. Application-aware byzantine fault tolerance. Proceedings of the 12th IEEE international conference on dependable, autonomic and secure computing; New York: IEEE; 2014. p. 45-50.
24. Zhao, W. Building dependable distributed systems. Wiley-Scrivener; New York: 2014.
25. Zhao, W. Int J Parall Emerg Distrib Syst. 2015. Optimistic byzantine fault tolerance; p. 1-14. (preprint)
26. Zhao W. Performance optimization for state machine replication based on application semantics: a review. J Syst Softw. 2016; 112:96–109.
27. Zhao, W., Babi, M. Byzantine fault tolerant collaborative editing. Proceedings of the IET international conference on information and communications technologies; UK: IET; 2013. p. 233-240.
28. Zhao W, Melliar-Smith PM, Moser LE. Low latency fault tolerance system. Comput J. 2013; 56(6): 716–740.
29. Zhao W, Moser LE, Melliar-Smith PM. Unification of transactions and replication in three-tier architectures based on CORBA. IEEE Trans Depend Secure Comput. 2005; 2(1):20–33.
30. Zhao W, Zhang H, Chai H. A lightweight fault tolerance framework for web services. Web Intell Agent Syst Int J. 2009; 7(3):255–268. (2009).
31. Zhao, W., Zhang, H., Luo, X., Zhu, Y. Enable concurrent Byzantine fault tolerance computing with software transactional memory. Proceedings of the 8th international conference on advanced software engineering & its applications; New York: IEEE; 2015. p. 67-72.

Fig. 1.

An example scenario where a transaction T_j is aborted and retried due to a conflicting read operation issued by transaction T_i , where $i < j$

Fig. 2.
Data structure maintained for multiversion concurrency control

Fig. 3.
Read–write conflict and write–read conflict in multiversion concurrency control

Fig. 4. Architecture of the research prototype for state-machine replication based on software transactional memory

Fig. 5.
Average throughput using ordered strong strict 2PL

Fig. 6. Average lock wait time and commit wait time for ordered strong strict 2PL

Fig. 7. Throughput and abort rate for conventional multiversion concurrency control

Fig. 8.
Throughput and abort rate for speculative multiversion concurrency control

Fig. 9.
Speedup comparison with competing approaches