Hardware Support For Relaxed Concurrency Control In Transactional Memory

Utku Aydonat and Tarek S. Abdelrahman
Edward S. Rogers Sr. Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, ON, CANADA
e-mail: {uaydonat,tsa}@eecg.toronto.edu

***Abstract*—Today's transactional memory systems implement the two-phase-locking (2PL) algorithm which aborts transactions every time a conflict happens. 2PL is a simple algorithm that provides fast transactional operations. However, it limits concurrency in applications with high contention by increasing the rate of aborts. More relaxed algorithms that can commit conflicting transactions have recently been shown to provide better concurrency both in software and hardware. However, existing approaches for implementing such algorithms increase latencies of transactional operations, require complex hardware support and alter standard cache coherence protocols. In this paper, we discuss how a relaxed concurrency control algorithm can be efficiently implemented in hardware. More specifically, we use a technique which approximates conflict-serializability and implement it in hardware on top a base hardware transactional memory system that provides support for isolation and conflict detection. Our novel hardware scheme is based on recording conflicts as they occur, instead of aborting transactions. Transactions serialize at commit time according to these conflicts by sending broadcast messages. Our evaluation of this hardware scheme using a simulator and standard benchmarks shows that it captures the benefits of conflict-serializability. Applications with long transactions and high contention benefit the most; abort rates are reduced up to 7.2 times and the performance is improved up to 66%. We argue that this improvement comes with little additional hardware complexity and requires no changes to the transactional programming model.**

***Keywords*-hardware transactional memory; serializability; synchronization; two-phase-locking; conflict-serializability**

I. Introduction

In the last decade, system designers have shifted towards the multi-core computing paradigm in order to keep up with the power and performance demands of the mainstream computing market. However, this new paradigm requires carefully designed multi-threaded programs in order to take advantage of the available hardware resources. This has increased the interest in systems that promise ease of parallel programming. In particular, Transactional Memory (TM) systems have gained considerable popularity in recent years. This is because TM promises to facilitate parallel programming by eliminating the need for user locks while delivering performance that is close to that of carefully designed fine-grain locks.

In essence, TM systems provide *atomicity*, *isolation* and *consistency* for shared-data accesses within critical sections, called *transactions* [1]. That is, (1) the effects of a transaction appear to be performed instantaneously all together, or none of them are performed, (2) data modified by a transaction cannot impact the state of others until the transaction completes (i.e., *commits*) and (3) the state of shared data is always consistent. Transactions achieve these properties by keeping track of shared-data reads and writes, which are referred to as *transactional actions*. Two actions are said to *conflict* if one transaction writes to the datum already accessed by the other.

Conflicts are a potential source for inconsistency for transactions. In general, TM systems implement the *serializability* model where the outcome of the concurrent execution matches the outcome of a serial execution [2]. In order to make sure that transactions are *serializable*, TM systems implement algorithms that determine what actions must be taken when a conflict occurs. These algorithms are called *concurrency control algorithms*. Existing TM systems (both hardware and software) implement the *Two-Phase-Locking (2PL)* algorithm [2]. In this algorithm, transactions are simply not allowed to perform conflicting actions. If a transaction attempts a conflicting access, the conflict is handled by aborting one of the conflicting transactions, or by delaying the access until one of the transactions commits. 2PL is a simple algorithm with straightforward implementations and, hence, it provides fast transactional actions. However, 2PL limits concurrency because every single conflict causes an abort or a delay even though some conflicting transactions can be serialized. This becomes a significant performance bottleneck for target applications that contain long-running transactions with a high degree of data sharing (i.e., high *contention*) [2], [3], [4].

Conflict-Serializability (CS) is an algorithm that is more relaxed than 2PL [5]. In this algorithm, transactions can perform conflicting accesses yet still commit successfully with no aborts or delays. This is achieved by keeping track of the order in which transactions perform their conflicting actions. The order of the actions imposes constraints on the order among transactions. If a serial order of the transactions can be found, it is said that the set of all actions performed by all the transactions (called the *schedule* of transactions) is *conflict-serializable*.

Conflict-serializability has been shown to improve the per-

formance for both software and hardware TM (HTM) systems by reducing abort rates [3], [6], [7]. However, the hardware implementations to date require complex architectural support with significant changes to the standard cache coherence protocols, cause overheads by enforcing commit order on transactions, and rely on word-based conflict detection and complicated contention management policies for good performance. TM remains a novel approach. Therefore, such complexities and overheads will likely make processor and system designers reluctant to commit hardware resources or to change standard hardware components.

We believe that it is possible for TM implementations with relaxed concurrency control algorithms to be simple and yet still deliver good performance. In this paper, we show that CS can be efficiently implemented in a TM system using a technique previously shown to improve performance for software transactions [3]. This technique is based on serially ordering transactions as they execute by assigning them *Serializability Order Numbers (SONs)*. The TM system keeps track of the order of transactional actions and assigns SONs to transactions at commit time based on the order of their conflicting transactions. Transactions commit if they can be assigned a SON, otherwise they abort.

More specifically, we describe a novel hardware TM system that is based on this SON technique. The system requires little additional hardware complexity on top of existing fundamental TM components used for conflict detection and isolation. The additional hardware consists of a few simple components per processor: a small read history table, a set of registers to hold SONs and related data, and a set of flags for keeping track of conflicts. In addition, a table is maintained in virtual memory to keep track of the SONs of writer transactions for each shared address. The use of these components introduces some overheads that reflect cache misses due to the accesses to the virtual table and messages used for communicating SONs, commit events and conflicts among transactions. The simplicity of our approach stands in contrast to other approaches that involve more complex changes to hardware, in particular cache coherence protocols [7].

We design the additional hardware components, implement them in a TM simulator and evaluate their effectiveness. The evaluation shows that our HTM system outperforms the base HTM system that uses 2PL. Performance improves by 29% on average in nine benchmarks. Thus, our simple hardware extensions are able to reap the benefit of CS. Further, performance is not significantly worse than an ideal CS implementation that incurs no overheads. The difference in performance is 6.7% on average, which leads us to believe that our implementation is efficient and validates our approach compared to more complex approaches that deliver performance close to this ideal performance. Finally, the performance of our system is not significantly sensitive to the parameters of the proposed hardware components (e.g., size of tables, bits used for hashing, etc.). Indeed, our evaluation shows that it is possible to select parameters that work well for a wide range of benchmarks.

The remainder of the paper is organized as follows. Section II gives a brief overview of the two concurrency control algorithms implemented in TM systems. Section III describes the SON method for approximating CS. Section IV presents the base hardware TM system. Section V describes the design and implementation of the additional hardware needed to support the use of SONs for relaxed concurrency. Section VI presents our experimental evaluation. Finally, Section VII overviews related work and Section VIII gives concluding remarks.

II. Concurrency Control in TM Systems

TM systems provide consistency of shared data by implementing the *serializability* model, i.e., they guarantee that the outcome of concurrent transactional actions is the same as if transactions executed in some serial order without any interference [2]. For this purpose, they implement concurrency control algorithms that determine how the actions of concurrent transactions can be ordered to match an equivalent serial execution.

Two-Phase Locking (2PL) is a strict algorithm that does not allow conflicting accesses while a transaction is running. If a conflicting access occurs, either one of the conflicting transactions is aborted or the conflicting access is delayed. This is illustrated in Figure 1. The figure shows an example schedule of three simple transactions TX1, TX2 and TX3 that access an address A. In the schedule shown, transactions conflict over A. Therefore, in any TM system that implements 2PL, the schedule shown cannot execute without aborts or without delaying accesses. This is regardless of which conflict-detection, contention management or version management algorithms are used. For instance, a TM system that uses eager conflict detection (eager CD) and requester-wins policy will abort TX1 as soon TX2 writes to A and will abort TX2 as soon as TX3 writes to A; a TM system that uses eager CD and requester-stalls will stall TX2's write action until TX1 commits and will stall TX3's write action until TX2 commits. Further, a TM system that uses lazy CD will again abort TX1 and TX2, only waiting until commit.

Conflict-Serializability (CS) is a more relaxed algorithm than 2PL [5]. Two schedules are *conflict-equivalent* if: (1) they contain exactly the same actions and (2) the order of the conflicting actions is the same. A schedule is said to be conflict-serializable if it is conflict-equivalent to some legal serial schedule. That is, we can find a valid serial ordering of transactions such that the order of their conflicting accesses remain the same [5]. The schedule in Figure 1 is conflict-serializable because TX1 $\rightarrow$ TX2 $\rightarrow$ TX3 is a valid ordering of transactions with respect to the order of their conflicting accesses[1]. In order words, this concurrent execution produces

[1]In a lazy system, the serialization will be TX1 $\rightarrow$ TX3 $\rightarrow$ TX2, because writes take effect at commit.

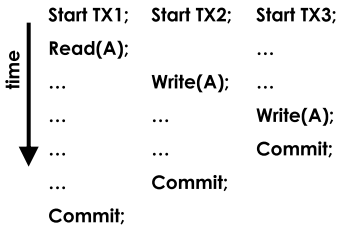


Figure 1. An example schedule for three transactions.

the same outcome as if TX1, TX2 and TX3 executed serially in this order . Therefore, unlike 2PL, a TM system that implements CS will commit all three transactions successfully without any delays or aborts.

III. Serializability Order Numbers

In earlier work [3], we propose a method for efficiently implementing conflict-serializability in a software transactional memory system. The method attempts to incrementally construct a conflict-equivalent serial schedule based on the actions of transactions. If such a schedule can be constructed then the transactions are serializable. The conflict-equivalent serial schedule is constructed by determining a *serializability order number* (SON) for each transaction. The SON is an integer that indicates the relative order of a transaction among all transactions in the conflict-equivalent serial schedule that is being constructed. The SONs of transactions are determined based on the relative order of their conflicting actions. That is, the transaction that performs its access first will have a smaller SON because, in any conflict-equivalent schedule, the relative order of the conflicting actions of transactions must be the same. If a unique SON can be determined for each transaction, then a conflict-equivalent serial schedule exists. Hence, it can assumed that all the read/write actions of transactions appear to have happened atomically and in the order of their SONs.

Whether an SON can be assigned to a transaction is not known until commit-time when all the conflicts of a particular transaction are known. Since allowing possibly non-serializable transactions to execute can cause side-effects (e.g., infinite loops, segmentation faults, etc.), it is desirable to keep transactions isolated until commit time. Hence, the SON method is typically used with a lazy version management (VM) system [8]. In such a system, writes take effect only when transactions commit and the commit order of transactions determines how transactions must be serialized. The SONs of transactions are determined using the following basic rules in a lazy VM system:

Forward-Serialization. If a transaction TX1 reads or commits (i.e., writes the value buffered into the memory) an address that has already been committed by another transaction TX2, then TX1's SON must be be higher than that of TX2. Similarly, if a transaction TX1 commits an address that was read by an already committed transaction TX2, then TX1's SON must be higher than that of TX2.

Backward-Serialization. If a TX2 commits an address that was already read by an active transaction TX1, then TX1's SON must be lower than that of TX2.

Conflict Type (Eager CD)	Load-After-Store	Store-After-Store	Store-After-Load
Schedule #	1	2	3
Schedule	Start TX1; ... Store(A); ... Commit; / Start TX2; ... ... Load(A); ... Commit;	Start TX1; ... Store(A); ... Commit; / Start TX2; ... ... StoreA); ... Commit;	Start TX1; ... Load(A); ... Commit; / Start TX2; ... ... StoreA); ... Commit;
Serialization	TX1 → TX2	TX1 → TX2	TX1 → TX2
Operation	TX2.LB = TX1.SON	TX2.LB = TX1.SON	TX2.LB = TX1.SON
Schedule #	4	5	6
Schedule	Start TX1; ... Store(A); ... ... Commit; / Start TX2; ... ... Load(A); ... Commit;	Start TX1; ... Store(A); ... ... Commit; / Start TX2; ... ... Store(A); ... Commit;	Start TX1; ... Load(A); ... ... Commit; / Start TX2; ... ... Store(A); ... Commit;
Serialization	TX2 → TX1	TX2 → TX1	TX1 → TX2
Operation	TX1.LB = TX2.SON	TX1.LB = TX2.SON	TX1.UB = TX2.SON
Schedule #	7	8	9
Schedule	Start TX1; ... Store(A); ... Commit; / Start TX2; ... ... Load(A); ... Commit;	Start TX1; ... Store(A); ... Commit; / Start TX2; ... ... Store(A); ... Commit;	Start TX1; ... LoadA); ... Commit; / Start TX2; ... ... Store(A); ... Commit;
Serialization	TX2 → TX1	TX1 → TX2	TX1 → TX2
Operation	TX2.UB = TX1.SON	TX2.LB = TX1.SON	TX2.LB = TX1.SON

Figure 2. All 9 possible conflicting schedules for two transactions under lazy VM.

Figure 2 shows all 9 possible conflicting schedules for two transactions in a TM system that uses lazy VM. The figure also shows how these two transactions can be serialized according to the above serialization rules. The serialization rules impose a lower bound (LB) and an upper bound (UB) on the SON of a transaction. The lower bound of a transaction is initialized to 0, and the forward serialization rules are applied to increase it. The upper bound is initialized to ∞ and the backward serialization rule is used to lower it. Figure 2 depicts how these rules are used to alter transaction lower and upper bounds. If at any moment during execution, the lower bound on the SON of a transaction becomes equal to or higher than its upper bound, this transaction cannot be placed in a conflict-equivalent serial schedule. In this case, the transaction aborts.

If a transaction performs all its accesses without aborting, it starts to commit and examines its SON range to determine a SON value. If the upper bound is not ∞, then it reflects the SON of some conflicting transaction. Thus, the SON of the transaction is selected as the upper bound minus one[2]. If the upper bound is ∞, then the SON of the transaction is set to the lower bound plus n, where n is the number of threads.

The forward serialization rule serializes transactions after already committed (reader or writer) transactions. For this purpose, when a transaction commits it assigns its SON to all the addresses it read and committed. The largest SON of all the transactions that committed an address is referred to as the *write-number* of the address. Similarly, the largest SON

[2]Integer numbers are used as SONs which impose the additional constraint that lower bound cannot be upper bound minus one.

of all the transactions that read an address is referred to as the *read-number* of the address. When a transaction accesses an address, its lower bound is increased to the write-number of the address (the lower bound remains the same if it is higher), effectively serializing the transaction after the last committer of the address. When a transaction commits an address, its lower bound is increased to the largest of the read-number and the write-number, effectively serializing the transaction after all the past readers and committers of the address.

IV. Base Hardware TM Support

In this work, we extend a base hardware EL system (i.e., *E*ager CD and *L*azy VM) and requester-wins policy[3]. We opt to use eager CD because it meshes well with cache coherence protocols on existing multiprocessor systems; conflicts are detected as memory accesses are issued using cache coherence messages. Similarly, we opt to use lazy VM because it allows the isolation of transactions until commit time when SON numbers are assigned to serialize the transactions. In contrast, eager VM systems need a mechanism to handle the side effects of non-serializable transactions until they abort at commit time, as mentioned above.

The EL base system is similar to that of Bobba et al. [8] and consist of unshaded components shown in Figure 3. It extends a typical multiprocessor architecture by adding: read and write sets for keeping track of reads and writes of transactions, a checkpoint register file for checkpointing registers at the start of transactions and a write-log for storing speculative writes of transactions. The architecture provides support for conflict detection, aborts and atomic commits [8].

The read and write sets are implemented using Bloom filters as opposed to cache line tags. The use of Bloom filters decouples conflict detection from cache states, as described in [9]. In our base system, we assume perfect bloom filters with no false positives, which ignores the impact of conflict detection mechanisms on performance. Further, transactionally stored data is kept in a FIFO write-log as directed by lazy VM.

Conflicts are detected by keeping track of cache coherence messages at the cache line granularity. When a processor receives a *get-shared* message for an address from a remote processor, it checks its write-set for possible load-after-store conflicts. If the write-set includes the requested address, a load-after-store conflict exists and the processor aborts the receiving transaction, as per the requester-wins policy. Similarly, when a processor receives an *invalidate* or a *get-exclusive* message for an address, it checks its read and write sets for possible store-after-store and store-after-load conflicts. If the either of read or write sets include the requested address, the receiving transaction aborts. In order to ensure that processors receive cache coherence messages, even after a speculatively loaded or stored line is evicted from the cache, the state of such lines is upgraded to a *sticky* state [9].

[3]Nested transactions and paging are not supported and each processor runs a single transaction at a time.

When a transaction aborts, it flushes its write-log, restores the register checkpoint, restores the program counter, clears its read and write sets and then restarts. We use an exponential backoff for transaction restarts. When a transaction commits, it iterates over its write set and commits its speculative stores into the memory. It then clears its read and write sets.

In addition the base system supports the use of commit tokens and the broadcast of NACK messages, both standard techniques and are necessary for our SON-based implementation, as will be described next.

The LL (i.e., *L*azy CD and *L*azy VM) configuration of the base system may outperform the EL configuration due to lazy detection of conflicts. This is typical of other systems such as FlexTM [10], TCC [11] and Bulk [12]. Thus, we also evaluate the performance of this LL configuration in Section VI. On the other hand, we expect the performance and the abort rates of the EE (i.e., *E*ager CD and *E*ager VM) configuration to be similar to our base system.

V. SONTM

In this section we describe the hardware additions needed to implement the SON-based concurrency algorithm. We refer to the system with the hardware additions as SONTM. First, we describe the general requirements imposed by the use of SONs to implement CS. We then describe the hardware components that implement these requirements.

A. System Requirements

SONTM aims to serialize transactions according to the order of their conflicting accesses, using SON lower and upper bounds. Thus, the main challenge is to able to handle every possible schedule for any two conflicting transactions. In our base system, which uses lazy VM, two transactions can conflict in 9 different schedules, as illustrated in Figure 2. The serializations in schedules 1-3 are handled by updating the lower bound of the active transaction (TX2 in the figure) with the write-numbers and the read-numbers of the addresses accessed by the two conflicting transactions. Hence, SONTM must implement a write-number table and a read-number table to keep the write-numbers and read-numbers for shared addresses.

In contrast, schedules 4-9 involve two conflicting active transactions. The order in which these transactions commit determines how they will be serialized. When one of the conflicting transactions commits, the other transaction's lower or upper bound must be updated with the SON of the committing transaction. Thus, SONTM employs a broadcast mechanism that is used by the committing transaction to send its SON to all the other transactions in the system. Conflicts are only recorded by the transactions that perform the conflicting access last (i.e., destination transactions of the conflicts). For this purpose, each transaction keeps a set of conflict flags to keep track of the type and the id of the rival processor for each of its conflicts. A transaction simply records the conflict in these flags if it receives a NACK

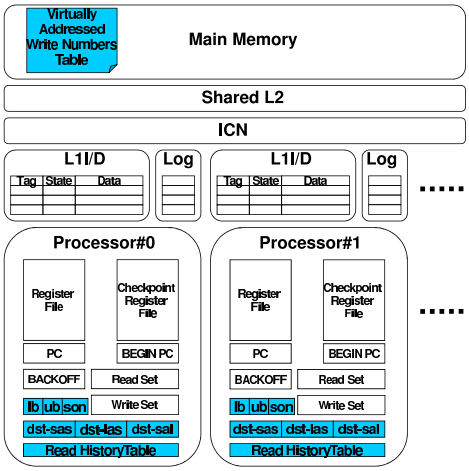


Figure 3. Hardware support overview.

message sent by a rival processor in response to a conflicting access. For a particular conflict, if the source transaction (i.e., performs the access first) of the conflict commits first, the destination transaction simply checks its conflict flags and updates its SON bounds with the SON broadcasted. On the other hand, if the destination transaction of the conflict commits first, the source transaction must update its SON bounds. However, because the source transaction does not have any records of the conflict, the SON-broadcast message of the destination transaction also includes two bit vectors that inform the receiving processors if they should update their lower or upper bounds with the SON included in the broadcast message.

B. Hardware Components

The key components of SONTM are the shaded components shown in Figure 3. They consist of *SON registers*, the *read history table*, and the *conflict flags* in each processor and the *virtual write-number table* in the virtual memory.

1) SON Registers: Two special purpose registers *lb* and *ub* are used to store the SON lower and upper bounds of the current active transaction. The *lb* register is initialized to 0 and can only increase. The *ub* register is initialized to the largest possible value (0xffffffff for a 32-bit system) and can only decrease. In addition, the *son* register stores the SON of the current transaction at commit time.

2) The Virtual Write-Number Table: It is necessary to keep write-numbers for each shared address in order to serialize transactions after committed writers (Schedules 1-2 in Figure 2). Since transactions in lazy VM TM systems store each address in their write-sets into the memory at commit time, it is convenient to keep the write-numbers of these addresses also in the memory, which results in simpler hardware design. Therefore, we use a write-number table in the virtual address space at a location known to all processors and its entries are initialized to 0. Virtual space addresses are hashed into the table using a subset of the address bits.

Write-numbers are stored into the table at commit time. When the committing transaction iterates over the write-log

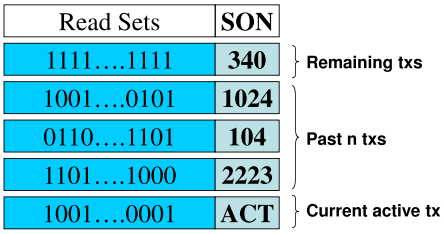


Figure 4. History table.

to commit the new values into the memory, it also calculates the virtual address where the write-number is located in the table for each address in its write-log. While the write-log is flushed into the memory, the write-numbers are also updated. Write-numbers are loaded when a load instruction is issued and during write-set validation. When a cache-line address is accessed, the virtual address of the associated write-number is calculated and loaded from the memory. This write-number is then used to update the *lb* register.

Loading and storing write-numbers in the table cause overheads due to accesses to the memory. These overheads can be minimized by overlapping the write-number accesses with the transactional data accesses. Further, since write-numbers are kept in the memory, they pollute the caches leading to cache misses. In Section V-D, we describe an optimization that eliminates some of the requests for write-numbers. We take into account all these factors when we evaluate the performance of the SONTM in Section VI.

Due to the use of hashing, it is possible for more than one cache line address to map to the same table entry resulting in aliasing. However, it is important to realize that the aliasing does not lead to incorrect execution but only to unnecessary aborts. For instance, a transaction TX will always be forward-serialized after a past transaction TXa if it accesses an address `a` which was committed by TXa. However, if `a` aliases with another address `b` in the table, TX may also serialize after TXb that committed `b` in the past, causing possibly an unnecessary abort. We evaluate the impact of the size of the table and the address bits used for hashing and our results show that these parameters have little impact on performance.

In addition, it is possible for a race condition over a table entry to exist between a committing transaction and transactions that are trying to read the same entry in the table. The order of accesses to the table determines how these transactions will be serialized. We expect the hardware commit operations to be fast and hence these race conditions to be rare. Thus, we simply opt to abort a transaction when it conflicts with a committing transaction, which is the only scenario under which such race conditions can occur.

3) The Read History Table: The SON-based method requires read-numbers to be kept for each shared address in order to serialize writer transactions after committed readers (Schedule 3 in Figure 2). Similar to how write-numbers are stored in a write-numbers table, it is desirable to keep a read-numbers table where each memory address is associated with a read-number. However, this approach incurs significant

overheads because it requires iterating over the read-set of a transaction at commit time in order to update the read-numbers in the table[4]. Thus, rather than keeping a global read-numbers table, we opt to distribute the read-numbers over processors, using *history tables*, as shown in Figure 3. Each history table entry (i.e., *a history*) contains two fields for a specific transaction: the read-set and the SON of that transaction. Because it is not possible to keep histories for all the transactions of a processor, we keep histories only for the last `n` successful transactions of a processor. To represent the remaining past transactions, a *summary* history is kept. The read-set of this summary history matches any address and contains the largest SON of all the transactions whose histories are not kept. We implement the read-sets in the histories also as perfect bloom filters; L. Yen et al. [9] shows that using simple filters of 2 KB delivers close to perfect conflict detection. Figure 4 shows the history table.

The read-set of an active transaction and its SON are stored as the most recent entry in the history table when the transaction commits. The remaining histories are shifted; the oldest entry is used to update the SON of the summary entry and then is discarded.

The histories are accessed during the write-set validation of a transaction. For each address in its write-log, the committing transaction sends a *read-number request* to all the processors and waits until it receives all the responses. Each processor that receives such a request looks up the requested address in its histories. The largest SON of all the histories that matches the request is sent back. If no match is found, the SON of the summary history is sent back. The requester processor updates its lower bound with the read-numbers sent from all the processors (the largest of these distributed read-numbers is the actual read-number of the address.)

Due to the limited number of histories kept and the inaccuracy of bloom-filters used, false conflicts are plausible where an SON value larger than the actual read-number of an address is sent back. These false conflicts may cause unnecessary aborts. However, because read-sets are only used for write-set validation at commit-time, the impact of these false conflicts is expected to be minimal.

4) Conflict Flags: Each processor keeps track of its conflicts in *dst* flags, which consist of bits that each records a conflict with a specific processor. They record the conflicts where the processor is the destination of the conflict (i.e., performs the conflicting access last). There are three *dst* flags: *dst-sal*, *dst-sas* and *dst-las* for each type of conflict respectively, store-after-load, store-after-store, and load-after-store.

Conflicts are detected by keeping track of cache coherence messages. When a processor receives a *get-shared* signal caused by a remote load instruction, it checks its write-set for a possible conflict. If a conflict is detected, it sends back a NACK-ST signal to the remote processor (the destination of the conflict) so that that the remote processor can record the conflict in its *dst-las* flag. Similarly, the read-set (write-set) is checked for possible conflicts when a processor receives an *invalidate* or a *get-exclusive* signal caused by a remote store instruction. The *dst-sal* (*dst-sas*) flag of the issuing processor is updated. Table I summarizes what signals are sent and the conflict flags that are set for each type of conflict.

[4]In contrast, transactions typically perform fewer write actions than read actions and the write-numbers are updated while the write-log is committed to memory according to the lazy VM.

Table I
FLAGS THAT ARE SET FOR EACH TYPE OF CONFLICT.

Received Signal	Condition	Conflict Type	Operation
get-shared	write-set matches	Load-After-Store	NACK-ST sent
invalidate	read-set matches	Store-After-Load	NACK-LD sent
invalidate	write-set matches	Store-After-Store	NACK-ST sent
get-exclusive	read-set-matches	Store-After-Load	NACK-LD sent
get-exclusive	write-set-matches	Store-After-Store	NACK-ST sent
NACK-ST	issued a load	Load-After-Store	*dst-las* set
NACK-LD	issued a store	Store-After-Load	*dst-sal* set
NACK-ST	issued a store	Store-After-Store	*dst-sas* set

C. Transactional Actions

In this section, we describe transactional actions. The steps taken for each action are shown in Figure 5.

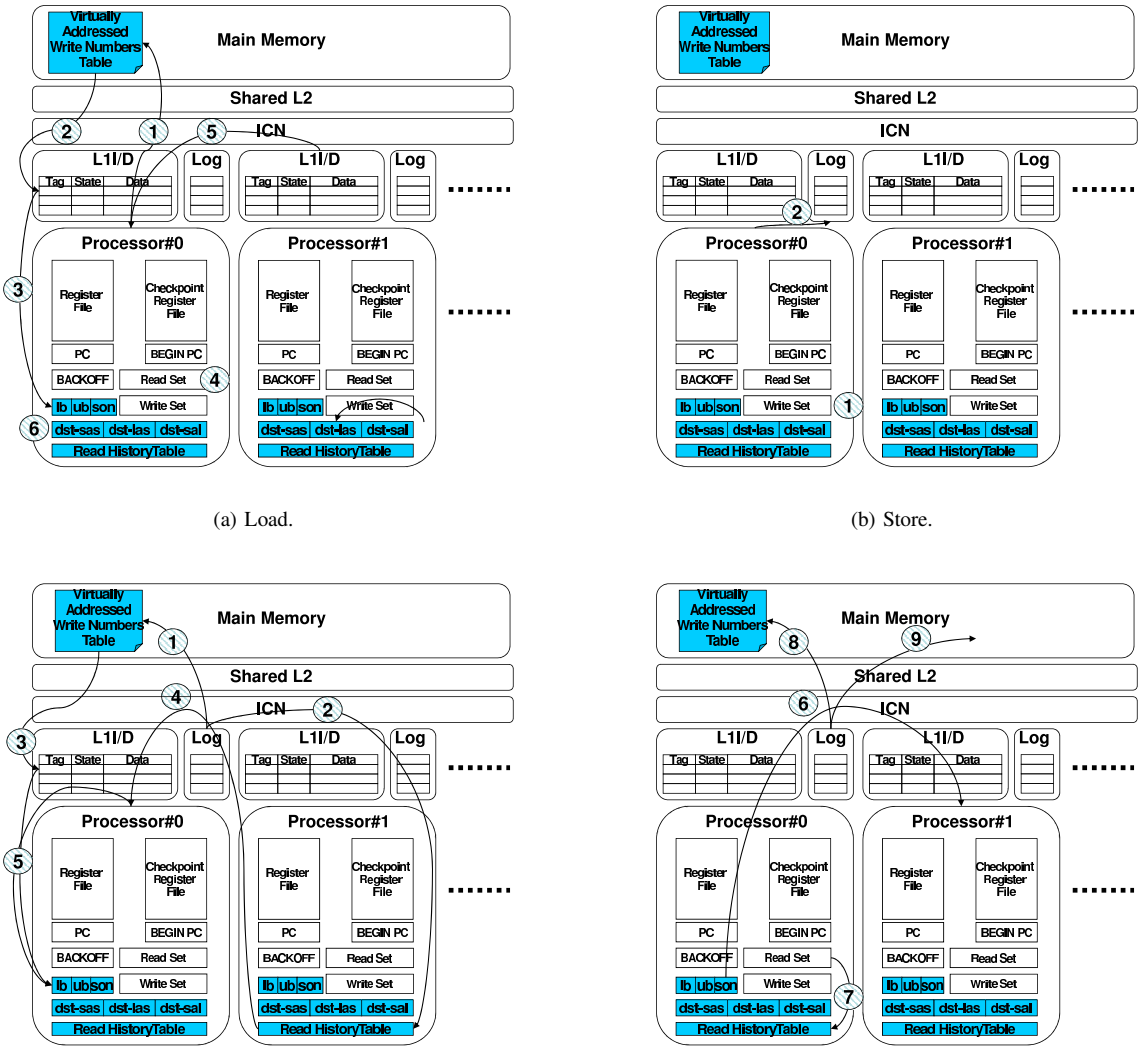
TX_BEGIN. The *lb* and *ub* registers are initialized to 0's and 0xffffffff's respectively.

TX_LOAD(addr). The transaction issues a load for the write-number of the `addr` from the virtual write-number table (① in Figure 5(a)). When the write-number is received ②, the transaction updates its *lb* register with the write-number (if it is higher) ③ and asserts that $lb < ub - 1$; otherwise, it aborts. The read-set is also simultaneously updated to include the cache line address of `addr` ④.

If the transaction receives a NACK signal from a remote processor ⑤ in response to the load instruction it issued, this represents a load-after-store conflict. Thus, it records the conflict by setting the corresponding bit in its *dst-las* flag ⑥. This conflict will be handled (i.e., conflicting transactions will be serialized) when one the conflicting transactions commit.

The `TX_LOAD` cannot complete until the write-number request is serviced and the transaction is verified for serializability. Although this presents a potential source for performance bottleneck, the write-number request can be serviced simultaneously with the actual load from the target address. Further, an optimization that eliminates some of the write-number requests is described later in this section. Our experimental evaluation presented in Section VI shows that the impact of the write-number requests on performance is minimal.

TX_STORE(addr, val). Since we use a lazy VM system, stores have no impact on consistency until the transaction commits. Therefore, no serializability checks are performed when a store instruction is issued. The cache line address of `addr` is inserted into the write-set(① in Figure 5(b)), and `val` is stored into the write-log together with `addr` ②.



(a) Load.

(b) Store.

(c) Validation.

(d) Commit.

Figure 5. SONTM transactional actions.

TX_COMMIT. The transaction starts committing by acquiring a commit token. Then, the transaction validates its stores: it iterates over the write-log and it issues a write-number request(① in Figure 5(c)) and a read-number request for each address in the log ②. These are issued simultaneously to overlap memory access latencies. When the transaction receives a write-number ③ or a read-number ④ for a certain address, it updates its *lb* register ⑤ and asserts that $lb < ub - 1$; otherwise, it aborts immediately. The transaction waits in the validation phase until the write-numbers and the read-numbers for all the addresses in the write-log are received. Then, it starts the second phase of commit. First, it selects its SON and broadcasts it to all the other processors(⑥ in Figure 5(d)). It also checks its *dst* conflict flags and determines which transactions (the source transactions of its conflicts) should update their SON bounds depending on the types of the conflicts. It includes this information in the broadcast message with two bit vectors called *uptLower* and *uptUpper*. Table II shows how these bit vectors are generated using simple bit operations. Second, the committing processor saves its read summary in the history tables ⑦. Then, it iterates over the write-log once again. During this iteration, it overwrites the write-numbers in the virtual table ⑧ and it commits values into the memory ⑨. The commit token guarantees that the write-numbers have not changed since the transaction started its commit. The transaction releases the commit token when all the write-numbers have been updated. At this point, it also clears its read and write sets, flushes its write-log and clears its conflict flags. If a processor receives a SON-broadcast message from a committing processor, it checks its outstanding conflicts with this processor. Table III shows what action is taken for each recorded conflict type.

The above mentioned hardware scheme is complete; i.e., it serializes transactions according to their conflicting actions for every conflict type. More specifically, of all the 9 possible conflicting schedules of two transactions shown in Figure 2,

Table II
HOW *uptLower* AND *uptUpper* BIT VECTORS ARE SET AT COMMIT TIME.

Condition	Updated Bit Vector	Schedule (Figure 2)
*i*th bit in *dst-las* is set	*i*th bit in *uptLower* is set	Schedule 4
*i*th bit in *dst-sas* is set	*i*th bit in *uptLower* is set	Schedule 5
*i*th bit in *dst-sal* is set	*i*th bit in *uptUpper* is set	Schedule 6

Table III
ACTIONS TAKEN BY PROCESSOR *i* WHEN IT RECEIVES A SON BROADCAST MESSAGE FROM PROCESSOR *k*.

Matching Conflict Flag	Updated Register	Schedule (Figure 2)
*i*th bit is set in *uptLower*	*lb*	Schedule 4,5
*i*th bit is set in *uptUpper*	*ub*	Schedule 6
*k*th bit is set in *dst-las*	*ub*	Schedule 7
*k*th bit is set in *dst-sas*	*lb*	Schedule 8
*k*th bit is set in *dst-sal*	*lb*	Schedule 9

Schedules 4-9 are handled by the active transaction when the other transaction broadcasts its SON. Further, Schedules 1-3 are handled by updating the lower bounds using the write-numbers and read-numbers of accessed memory addresses.

Further, since SONTM builds upon a base HTM system, it is possible to adjust the concurrency control separately at each processor. Some transactions can use 2PL and abort when they receive conflicting coherence requests, while others executing on different processors use the SON-based method. This can be useful in two ways: in applications that contain transactions with different characteristics, the user can choose to execute the transactions with low contention with 2PL, disabling SONTM hardware components. Second, adaptive techniques can be implemented to decide which algorithm to use based on the characteristics of the target application. Such adaptive techniques will explored in future work.

D. Optimizing Transactional Actions

In the SONTM implementation described above, every `TX_LOAD` action loads from the virtual memory the write-number of the address. Further, during `TX_COMMIT` requests for write-numbers and read-numbers are sent to all the processors to validate stores. These operations incur overheads that increase the latencies of transactional loads and commits. However, as discussed below, write-numbers and read-numbers are not always required to validate transactional loads or stores. Thus, in some cases, requests for write-numbers and read-numbers can be eliminated, which reduces the overheads of the SONTM.

We say that a committed transaction is *retired* if it does not have an impact on serialization anymore. Retired transactions can safely be ignored by future transactions.

A committed transaction can retire if there are no active transactions serialized before it. This can be argued as follows. A transaction can retire when it is known that it cannot be included in a cycle in the serializability graph in the future. Whether or not a transaction can be included in a cycle can found as follows. When a transaction `TXC` commits successfully, we know that it is not part of a cycle. `TXC` will only be a part of a cycle in the future with the addition of a new edge in the serializability graph, caused by an access of an active transaction `TXA`. Because this access happens after `TXC` committed, `TXA` will be serialized after `TXC` in the serializability graph, i.e., `TXC`→...→`TXA`. If this new edge creates a cycle in the serializability graph, this means that `TXA` is also serialized before `TXC`, i.e., `TXA`→...`TXC`→...→`TXA`. This is only possible if `TXA` was in fact active and serialized before `TXC` when `TXC` committed. Thus, we can conclude that `TXC` can only be included in a cycle, if there exists a transaction such as `TXA`. In other words, `TXC` cannot anymore be included in a cycle after it commits, if they are no active transactions that are serialized before it.

Optimization. We can say that if none of the active transactions running in the system have their upper bounds updated (i.e., their upper bounds are still equal to ∞ initialized at start time), this means all the transactions that committed in past are retired. This is because there exists no active transaction serialized before any of the past transactions. Consequently, write-numbers and read-numbers kept up to this point are obsolete; a transaction does not need to update its lower bound neither with the write-number nor with the read-number of the address it is accesses.

Implementation. Transactions keep track of the upper bound updates that occur in the system. When a transaction is about to issue a request for the write-number or the read-number of an address, it checks if there are any active transactions with their upper bounds set. If this is not the case, then all the past transactions are retired, and hence, the transaction does not issue the request for the write-number or the read-number.

Table IV
HARDWARE MODEL PARAMETERS.

Component	Parameters
Processors	8-core CMP, 5 GHz in-order single-issue Sparc processors
Interconnect	packet-switched, 2 clusters with 4 cores in each
L1 Cache	MESI protocol, 32 KB private, write-back, 1-cycle latency
L2 Cache	Shared 8 MB, 32 banks, 34-cycles latency
Directory	on-chip directory with 6-cycles latency
Memory	4 GB, 500-cycles latency

VI. EXPERIMENTAL EVALUATION

We build SONTM using a simulator of the EL hardware TM implementation described in [8]. This system provides the base support for TM as described in Section IV. It is built using the Simics [13] full-system simulation infrastructure. The Wisconsin GEMS toolset [14] provides support for customizing memory models. Simics accurately models the SPARC architecture, with in-order single-issue processors. The TM support is implemented using magic instructions, i.e., special instructions caught by Simics and passed on to the memory model. The simulated target system runs Solaris 10 to provide OS support for applications.

Table IV shows the parameters of our simulations. The simulations take into account at the cycle level the overheads of our SONTM implementation by keeping track of contention over the interconnect, cache effects, virtual memory, as well as the latencies of accessing the cache, its directory and the memory system. These overheads reflect accesses to

the virtual write-number table in the main memory, broadcast message latencies, stalls due to NACK responses and read-number requests and their responses.

A. Evaluated HTM Systems

`2pl-EL` is the standard EL hardware transactional system used as the base system to implement SONTM. It is based on the 2PL concurrency algorithm, where each conflict causes an abort. `2pl-LL` is an LL configuration of the hardware TM system used by Bobba et al. [8]. It only differs from `2pl-EL` in that the conflicts are handled at commit time in accordance with lazy conflict detection. We omit the evaluation of a `2pl-EE` system because it incurs abort rates close to those of `2pl-EL` as pointed out in Section IV.

`datm` implements the commit ordering concurrency model used by DATM [7], another TM proposal that implements conflict-serializability in hardware. In our implementation, (1) conflicts are detected eagerly, (2) transactions commit in the order of their conflicting actions, (3) when a load-after-store conflict is detected, data is forwarded from the writer transaction to the reader transaction, hence, the reader transaction is not stalled, and (4) when a transaction aborts, all the other transactions that it forwarded data to are also aborted. Unlike DATM, in this HTM implementation conflicts are detected at the cache-line granularity for fair comparison with our SONTM. Furthermore, this is an ideal implementation of the concurrency model of DATM. That is, the latencies of control messages that are necessary to keep track of conflicts, race conditions, and data forwarding are not taken into account.

`sontm-ideal` is an ideal version of our SONTM hardware transactional memory system which uses the SON-based method to provide consistency of shared-data (i.e., the serializability property). It is ideal in the sense that it does not incur any additional overheads due to the implementation of the SON-based technique. That is, accessing SON-tables, updating SON lower and upper bounds, and serializability checks are implemented as no-overhead operations on top of the base hardware. Therefore, we use this ideal implementation to evaluate the maximum performance gain that can be achieved using the SON-based technique.

`sontm` is the SONTM hardware transactional memory system described in Section V with all its features and their associated performance penalties. The size of the virtual table is 64Kb. Bits 8-23 are used to map an address to an entry in the table. A 4-entry read-set history table is kept for past transactions at each processor, in addition to one entry for the remaining past transactions and the read-set of the active transaction. Responses to the read-number requests are sent similar to a memory data response, first to the L2 cache and then to the target processor. The broadcast of SON-numbers is also implemented similarly.

B. Benchmarks

We assess the impact of the SON-based method on performance using the set of benchmarks that consist of an ordered linkedlist program and the STAMP benchmarks [15]. We use the same parameters used in earlier work for evaluating other HTM implementations [7]. These benchmarks contain both short-running low-contention transactions as well as long-running high-contention transactions, thus, demonstrating the pros and cons of our scheme.

Table V shows for each benchmark run on the base EL system with 8 threads: the parameters of the benchmark, the ratio of cycles spent inside transactions to the total execution cycles (Xact Ratio), the total number of committed transactions (Total Xacts), the average cycles spend for each successful transaction (Xact Cycles), the abort rate, the ratio of wasted cycles (Wasted Cycles), the average number of read actions (Xact Reads), the average number of write actions (Xact Writes) and concurrency. The abort rate is measured as the ratio of the number of aborted transactions to the total number of transactions committed. The wasted cycles is defined as the total number of cycles spent by transactions that later aborted plus the number of cycles spent in the exponential backoff until restart. The table shows the ratio of wasted cycles to the total execution cycles. Concurrency is the ratio of the total execution cycles with 1 thread to the execution cycles of the same benchmark with 8 threads. From the table, it can be seen that `list`, `bayes`, `labyrinth`, and `yada` have longer transactions; `list`, `bayes`, `vacation`, `labyrinth` and `yada` have higher abort rates (0.96 and higher) and, hence, higher wasted cycles ratios.

C. The impact of the SON Support

Figure 6(a) shows the total execution cycles of each HTM system for each benchmark normalized with respect to the total execution cycles of the `2pl-EL` HTM system. A number of observations can be made from the figure. First, the total execution cycles of the HTM systems correlate with the abort rates shown in Figure 6(b) and with the wasted cycles shown in Figure 6(c). This shows that reducing the abort rates of a benchmark positively impacts its performance by reducing the cycles wasted for aborted transactions. Second, the SONTM versions (i.e., `sontm-ideal`, and `sontm`) outperform `2pl-EL` for the benchmarks with high abort rates (i.e., `list`, `bayes`, `vacation`, `labyrinth`, and `yada`) More specifically, `sontm` improves the performance by 34% for `list`, 32% for `bayes`, 66% for vacation, 57% for labyrinth and 93% for yada. This result is expected since abort rates are significantly reduced for these benchmarks. For `intruder` and `genome`, smaller gains are obtained because these benchmarks have lower abort rates under 2PL. The `kmeans` benchmark contains transactions that read and write to the same address, and, hence, are not conflict-serializable when they conflict. Thus, its performance does not improve. The third observation is that the performance of `sontm-ideal` is close to that of `datm`. This shows that the SON algorithm is an efficient method for implementing CS in a hardware TM system. In fact, in some benchmarks (`intruder`, `genome`, `kmeans` and `ssca2`), `sontm-ideal`

Table V
BENCHMARK PARAMETERS AND CHARACTERISTICS.

Benchmark	Parameters	Xact Ratio	Total Xacts	Xact Cycles	Abort Rate	Wasted Cycles	Xact Reads	Xact Writes	Concurrency
list	*-init 4096 -range 0:8192 -mix 1:1:1*	94.9%	8,192	99,311	0.96	46.0%	2,080.1	0.5	4.01
bayes	*-v 32 -r 384 -n 2 -p 20 -s 1*	62.7%	797	344,889	1.39	50.7%	10,363.1	1,588.0	2.03
vacation	*-t 20000 -n 10*	56.5%	20,000	5,899	1.01	49.1%	651.3	36.5	1.81
labyrinth	*-i random-x48-y48-z3-n48.txt*	93.2%	112	923,514	5.89	88.2%	88,451.5	11,356.7	0.41
yada	*-a20 -i 633.2 -t 8*	99.4%	5,080	26,430	4.96	94.5%	1,269.5	222.8	0.46
intruder	*-a10 -l4 -n2048 -s1 -t8*	57.0%	11,267	834	0.31	33.7%	28.5	6.5	4.92
genome	*-g 512 -s 16 -n 16384*	74.2%	10,785	3,040	0.08	18.0%	177.1	5.9	5.24
kmeans	*-m15 -n15 -t0.05 -irandom-n16384-d24-c16*	11.5%	43,706	396	0.06	2.3%	40.0	19.0	5.10
ssca2	*-s 13 -i 1.0 -u 1.0 -l 3 -p 3*	7.8%	47,277	298	0.01	0.4%	3.0	2.0	4.47

slightly outperforms datm due to stalls incurred by commit ordering of transactions. Fourth, the difference in performance between sontm-ideal and sontm is small (except for intruder and ssca2). Thus, it can be concluded that the impact of hardware components of sontm (write numbers table, read histories, NACK messages, broadcast messages) on performance is also small. Also, the performance impact of the limited size of write numbers table and the accesses to the table is small. Firth, the performance of sontm is close to that of an ideal (i.e., incurs no overheads) of datm, even though sontm has simpler implementation that requires no changes to the cache coherence protocols. Sixth, 2pl-LL outperforms 2pl-EL in vacation, labyrinth and yada due to its lower abort rates. However, the performance of 2pl-LL is still significantly lower than that of sontm because of the strict 2PL concurrency algorithm it implements. Finally, for ssca2, we observe that the performance is reduced by 14% when run with sontm compared to 2pl-EL. This is because this benchmark presents the worst case scenario for sontm; large number of short transactions are executed and the abort rate is close to zero. Thus, no performance gain is obtained with relaxed concurrency, yet frequent accesses to the write-numbers table increase cache misses, and lower the performance.

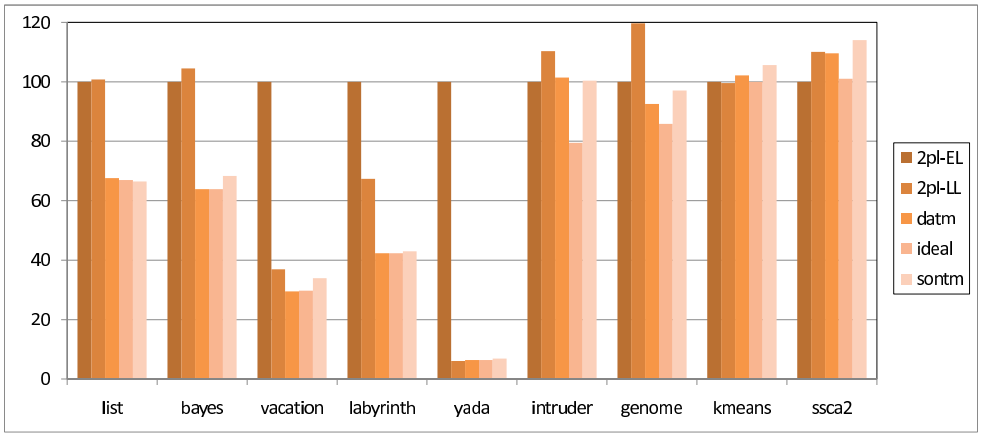
Figure 6(b) shows the abort rates of each HTM system evaluated for the benchmarks with long transactions and high contention (list, bayes, vacation, labyrinth, yada and intruder). The figure shows that the systems that implement CS have lower abort rates in each benchmark. For instance, the abort rate of the list benchmark is reduced from 0.96 with 2pl-EL to under 0.15 with son-ideal. This result confirms earlier work [3], [4], [7] that CS reduces abort rates in TM systems. Further, the figure shows that the abort rates of the systems that use the SON method (sontm-ideal and sontm) are also small and significantly lower than that of 2pl-EL. This demonstrates that abort rates can also be reduced significantly by implementing the SON method. Also, the difference in abort rates between sontm-ideal and sontm is generally small. This leads us to conclude that impact of the size of write number table and the aliasing it causes is not significant. For bayes, aliasing and overheads of accessing the write-numbers table impact the performance as these benchmarks have long transactions with large number of shared data accesses. The abort rates of genome, kmeans and ssca2, which are not shown in the figure, are already low with 2pl-EL system (0.08, 0.06, and 0.004 respectively). Thus, we do not observe much performance improvement when this benchmark is run under CS. Finally, 2pl-LL incurs lower abort rates compared to 2pl-EL due to the lazy handling of conflicts. However, the abort rates of 2pl-LL are still significantly higher than the abort rates of the systems that implement CS.

Figure 6(c) shows the wasted cycles normalized with respect to the total execution cycles of 2pl-EL for the benchmarks with high abort rates. Due to high contention and long transactions, wasted cycles are high for these benchmarks. We also see that the implementation of CS with the SON based method reduces the wasted cycles. This directly correlates with the reduction in the abort rates as shown in Figure 6(b).

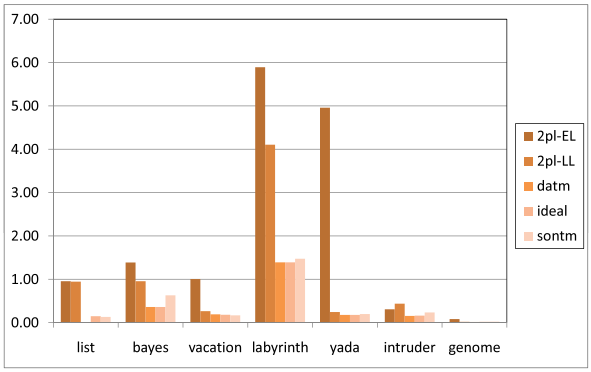
D. The impact of SONTM parameters

Figure 7(a) shows the total execution cycles of sontm normalized with respect to the total execution cycles of 2pl for the benchmarks. In each experiment, a different range of address bits is used for hashing. The size of the write-numbers table is constant at 64K. It can be seen that the address bits selection can impact the performance in some benchmarks. This impact may be as large as 31% of total execution cycles in list and bayes, or as small as 3% in vacation, 3% in labyrinth, 0.1% in yada and 5% in intruder. This mostly depends on how the data structures are aligned with cache lines in benchmarks. However, it is possible to use address bits starting from the 6th, 8th or 10th bits and generally produce good performance.

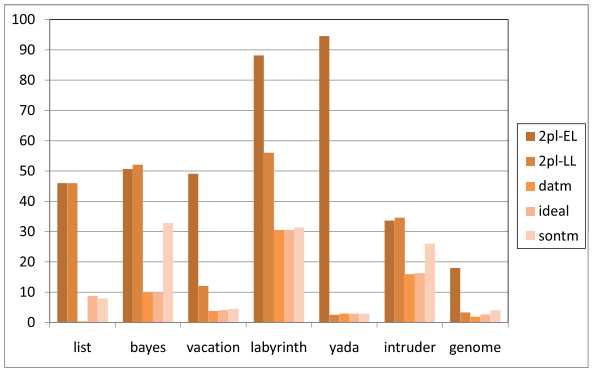
Figure 7(b) shows the total execution cycles of sontm normalized with respect to the total execution cycles of 2pl-EL for the benchmarks. We simulate execution with write-numbers tables of sizes 4KB, 16KB, 32KB, 48KB and 64KB. In each experiment, the 16 address bits starting from the 8th bit are used to hash into the table. It can be seen that the impact of the size of the table is small expect for list and bayes. This is because transactions in these benchmarks access a large number of shared addresses requiring a larger write-numbers table to eliminate the impact of aliasing. Nonetheless, the table size of 64K is generally adequate.



(a) Total execution cycles normalized to `2pl-EL`.



(b) Abort rates.



(c) Wasted cycles normalized to `2pl-EL`.

Figure 6. Normalized total execution cycles, abort rates and wasted cycles on 8 processors.

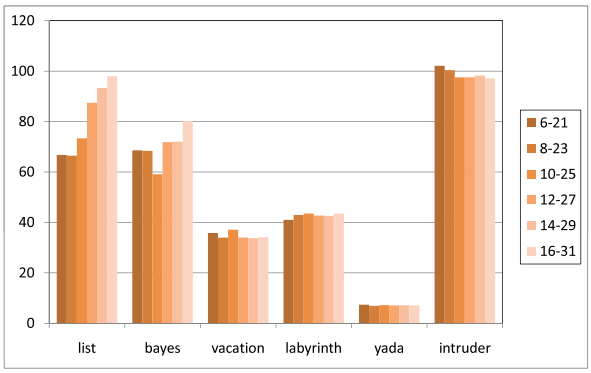
VII. Related Work

CS is a well-known concept in databases [5]. Ramadan et al. [7] implement CS in their DATM hardware TM system. They require new cache coherence protocols, which results in a complex hardware implementation. Given that future HTM systems will likely be based on existing coherence protocols, our scheme presents a clear advantage. SONTM requires no changes to existing cache coherence protocols and only the addition of simple hardware components, such as conflict flags and read histories. In addition, SONTM achieves performance that is close to the best that can be obtained from a hardware CS implementation for the target applications. Thus, its performance in the worst case is only slightly lower than that of an ideal DATM but at lower hardware complexity. Further, DATM ensures serializability by committing transactions in the order of their dependences, which creates delays. In contrast, our SONTM does not impose any commit ordering. DATM's data forwarding also relaxes the isolation properties of transactions leading to cascading aborts and inconsistent state in reader transactions. SONTM does not suffer from cascading aborts and does not allow inconsistent state in transactions. DATM also requires additional hardware support in cache controller and relies on word-based conflict detection and on contention management policies for achieving good performance. In contrast, SONTM uses simpler cache line based conflict detection and does not rely on contention management.

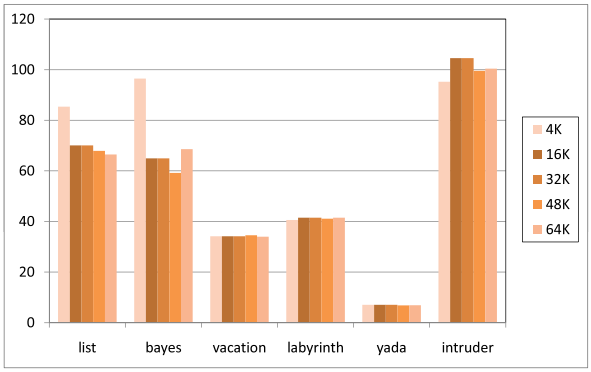
In earlier work [3], we describe how CS can be implemented using the SON-based technique in a software TM system. We demonstrate that CS can improve performance by lowering the abort rates, especially for applications with long running transactions and high contention. We also explore the feasibility of a hardware implementation [4]. In contrast, this work proposes concrete hardware components to implement the scheme and thoroughly evaluate it.

There have been several other approaches to relaxing concurrency in TM systems. Calstrom et al. [2] propose transactional collection classes to reduce the number of conflicts. However their approach requires semantic knowledge of data structures and their dependences. Riegel et al. [16], [17] investigate the use of snapshot isolation. However, snapshot isolation does not provide linearizability and hence requires the programmer to make sure that it is applicable.

Thread Level Speculation (TLS) [18] is a concept similar to TM. However, TLS parallelizes program constructs whereas TM provides mutual exclusion for critical sections. While epochs in TLS are inherently ordered, SONTM transactions



(a) Normalized execution cycles vs. address bits.



(b) Normalized execution cycles vs. virtual table size.

Figure 7. Impact of the address bits and the size of the virtual table.

are not ordered like in any other TM system. The SONs are only used to verify the serializability property of transactions.

VIII. Conclusions

Current TM system designs implement the Two-Phase-Locking (2PL) concurrency control algorithm which aborts or delays transactions when a conflict occurs. This paper presents a novel design of a hardware TM system that implements the conflict-serializability (CS) algorithm. This algorithm increases concurrency of transactions by allowing conflicting transactions to commit successfully, if a serial ordering of transactions can be found. Our system is unique because it uses serializability order numbers in hardware to serialize transactions transparently to the user. This results in simpler hardware design that does not require any changes to the standard cache coherence protocols or complex hardware support. The additional hardware support required consists of only small read histories and flags to keep track of conflicts in each processor, a table in virtual memory, and broadcast and NACK messages. Experimental evaluation shows that our system improves the performance of applications by 29% on average compared to a base system that uses 2PL. Further, the performance of our system is comparable to that of a CS implementation that incurs no overheads, which shows that our implementation is efficient. Finally, we show that the parameters of the implementation do not impact the performance significantly in most benchmarks and it is in fact possible to select parameters that work well for all benchmarks.

Our work can be extended to address the overheads of SONTM for applications that have low contention and short transactions such as ssca2, by using adaptive techniques. A simple algorithm can detect that an application only consists of short transactions that rarely conflict and switch the concurrency algorithm to 2PL.

References

[1] M. Herlihy and J. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proc. of ISCA*, 1993, pp. 289–300.

[2] B. Carlstrom et al., "Transactional collection classes," in *In Proc. of PPoPP*, 2007, pp. 56–67.

[3] U. Aydonat and T. Abdelrahman, "Serializability of transactions in software transactional memory," in *TRANSACT: Workshop on Transactional Computing*, February 2008.

[4] U. Aydonat and T. Abdelrahman, "Hardware support for serializable transactions: A study of feasibility and performance," in *TRANSACT: Workshop on Transactional Computing*, 2009.

[5] T. Connolly and C. Begg, *Database Systems: A Practical Approach to Design, Implementation and Management*. Addison-Wesley, 2009.

[6] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel, "Committing conflicting transactions in an STM," in *Proc. of the Symp. on Principles and Practice of Parallel Programming*, 2009, pp. 163–172.

[7] H. Ramadan, C. J. Rossbach, and E. Witchel, "Dependence-aware transactional memory for increased concurrency," in *Proc. of Int'l. Symp. on Microarchitecture*, November 2008, pp. 246–257.

[8] J. Bobba et al., "Performance pathologies in hardware transactional memory," *Comput. Archit. News*, vol. 35, no. 2, pp. 81–91, 2007.

[9] L. Yen et al., "LogTM-SE: Decoupling hardware transactional memory from caches," in *HPCA '07*, 2007, pp. 261–272.

[10] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible decoupled transactional memory support," in *Proc. of the Int'l. Symp. on Computer Architecture*, 2008, pp. 139–150.

[11] 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," in *Proc. of the Int'l. Symp. on Computer Architecture*, 2004, p. 102.

[12] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk disambiguation of speculative threads in multiprocessors," in *Proc. of the Int'l. Symp. on Computer Architecture*, 2006, pp. 227–238.

[13] P. Magnusson et al., "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[14] M. Martin et al., "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[15] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *Proc. of Int'l. Symp. on Workload Characterization*, 2008.

[16] T. Riegel, C. Fetzer, and P. Felber, "Snapshot isolation for software transactional memory," in *TRANSACT: Workshop on Transactional Computing*, 2006.

[17] T. Riegel, P. Felber, and C. Fetzer, "A lazy snapshot algorithm with eager validation," in *Proc. of Int'l. Symp. on Distributed Computing*, 2006, vol. 4167, pp. 284–298.

[18] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 1–12.