Hardware Support For Serializable Transactions: A Study of Feasibility and Performance

Utku Aydonat Tarek S. Abdelrahman

Edward S. Rogers Sr. Department of
Electrical and Computer Engineering
University of Toronto
{uaydonat,tsa}@eecg.toronto.edu

Abstract

The conflict-serializability consistency model can reduce abort rates and improve performance in software transactional memory (TM) systems. We discuss how this model can be supported on hardware-based TM systems. We consider a hybrid implementation where serializability is implemented as a software layer on top of a base hardware TM system. We also extend the base hardware TM system to fully support serializability in hardware with minimal compiler support. We implement and evaluate the hybrid system and an ideal version of the full hardware system. Our evaluation using two typical benchmarks shows that support for serializability in hardware-based TM systems results in reduced conflicts. However, the overhead of software in the hybrid implementation eliminates any benefits; performance is 2-5 times slower compared to the base hardware system. In contrast, the ideal hardware system improves performance by up to 23.2%, because of the reduction in the aborts. We conclude that with efficient implementations in hardware, conflict-serializability can improve performance, especially for applications with high cost of aborts.

1. Introduction

The emergence of multi-cores into mainstream computing has given Transactional Memory (TM) systems considerable popularity. This is mainly because TM promises to facilitate parallel programming, resulting in performance that is comparable to that of fine-grain locking with the programming simplicity of coarse-grain locking. In essence, TM systems provide *atomicity*, *consistency* and *isolation* for the shared-data accesses within critical sections, called *transactions* [1]. Over the last few years, several software of hardware implementations of TM systems have been proposed, differing mainly in how they ensure that these criteria are met. In general, the design space of TM systems (whether implemented in software or in hardware) spans the space generated by various approaches to: (1) when conflicts between transactions are detected, (2) where the updates made by transactions are kept, (3) which transactions, in the case of conflicts, are aborted, and (4) how the consistency of reads is ensured. The exhaustive investigation of this design space resulted in both an understanding of trade-offs among approaches and in performance that is close to that of fine-grain locking for applications with short transactions and low degree of data sharing.

In contrast, TM systems fail to provide acceptable performance levels for applications with long-running transactions and high-degrees of data sharing [2]. This is mostly due to the fact that software (and indeed hardware) TM systems *imitate* a two-phase locking (2PL) model to ensure the consistency of shared-data[1]. Under 2PL, transactions are aborted or delayed, if they perform conflicting accesses. Although 2PL is simple to implement and does provide fast transactional operations, it may be overly-restrictive and may degrade performance by causing unnecessary aborts [2]. Consequently, we have been investigating consistency models as yet another dimension of the TM design space. More specifically, we have explored the use of conflict-serializability ($\mathcal{CS}$) instead of 2PL [2]. We demonstrated that the $\mathcal{CS}$ model can outperform 2PL in a software TM system for applications with long-running transactions and high data sharing, mainly because of the reduction in abort rates. Although the use of $\mathcal{CS}$ introduces overheads due to extra bookkeeping and consistency checks, the benefits outweigh the overheads for such applications.

The popularity and benefits of TM is leading many research groups to consider hardware systems that provide TM support. Furthermore, one commercial system Sun Microsystem's Rock Architecture has emerged recently [3]. This motivates us to consider systems and algorithms that take advantage of the TM features that are available now, or may commercially become available in the near future. In this paper, we investigate how to leverage such *base* hardware support for conflict detection and atomic commits to efficiently implement $\mathcal{CS}$. We first describe the support that must exist to realize $\mathcal{CS}$ in any hardware-based TM system. We then consider two approaches to implementing this support. First, we consider a hybrid approach in which $\mathcal{CS}$ is implemented in a software layer that utilizes the base support without modifying it. Second, we extend the base hardware support for conflict detection and version management to fully implement $\mathcal{CS}$ in hardware with minimal compiler support.

We evaluate the two approaches using a simulator of the LL (lazy version management, lazy conflict detection) hardware-based TM system proposed in [4]. We fully implement the hybrid approach on top of the simulator, which allows us to understand the benefit-performance tradeoffs that exist with this approach. We also extend the simulator with an ideal version of hardware that implements $\mathcal{CS}$ but incurs no hardware overhead due to $\mathcal{CS}$. This allows us to determine an upper bound on the performance gains that can be achieved by fully supporting $\mathcal{CS}$ in hardware.

Our evaluation shows that the use of $\mathcal{CS}$ reduces abort rates in both the hybrid and hardware implementations. However, in the hybrid implementation, the overheads of the $\mathcal{CS}$ software outweigh the benefits of reduced abort rates, reducing performance by a factor of 2-5 times compared to the base system. In contrast, the ideal hardware implementation improves application performance. For

[1] The actual implementation may not necessarily be lock-based and in fact may use optimistic concurrency.

a linked-list application, which has long-running transactions and high abort rates, the improvement over the base hardware system is significant, about 23%. However, for a red-black tree application, which has short-running transactions and low abort rates, the benefit is (as expected) minimal at about 1.4%. These results lead us to conclude that it is unlikely for a hybrid implementation to be viable in practice but encourage us to pursue a detailed hardware implementation that supports $\mathcal{CS}$.

The remainder of the paper is organized as follows. Section 2 reviews transactional memory, consistency models implemented by transactional memory systems, the design space of hardware transactional memory systems and our earlier software-based method for implementing $\mathcal{CS}$. Section 3 discuses how $\mathcal{CS}$ can be implemented using our method on an LL hardware-based TM system. More specifically, it discusses the type of support that is necessary and how this support can be realized in a software layer on top the LL system or fully in hardware. Section 4 presents our evaluation of these two implementations using simulation and representative benchmarks.

2. Background

In this section, we give a brief overview of transactional memory and the consistency models implemented by transaction memory systems. Further, we describe our previous work on using serializability order numbers (SONs) to implement the conflict-serializability consistency model in software TM systems.

2.1 Transactional Memory

A transaction is a sequence of instructions that access and modify shared data and that satisfies atomicity, consistency and isolation criteria. Respectively, these criteria imply that (1) the effects of transactions appear to be performed instantaneously all together, or none of them are performed, (2) the state of shared data is always consistent, and (3) data modified by a transaction cannot be seen by other transactions until the transaction successfully completes.

A transaction can perform a number of read and write actions between its start and commit. The *schedule* or the *history* of a set of transactions is the list the actions performed by transactions in the order they happen in real time. A *conflict* is said to exist between two actions if the actions belong to different transactions, these actions access the same address, and at least one of these actions is a write. A *serial schedule* is a schedule in which the actions of one transaction do not interleave with the actions of another. Two schedules are said to be *conflict-equivalent* if: (1) they contain exactly the same actions, and (2) the order of the conflicting actions is the same. Two schedules are said to be *view-equivalent* if: (1) they contain exactly the same actions, and (2) the results of the execution is the same (i.e. the same values are read and the final state of the system is the same).

Transaction memory (TM) systems guarantee that transactional actions executed by concurrent threads appear to be performed in some serial order without any interference. That is, the effects of these actions will be the same as if the transactions executed sequentially in some serial order. If the accesses of a transaction cannot be ordered properly, the transaction must be aborted.

2.2 Consistency Models

A consistency model determines how the actions of concurrent transactions can be ordered to match an equivalent sequential execution. Given a particular schedule, a more relaxed a consistency model can find a larger set of equivalent serial schedules. That is, transactions have a better chance of successfully committing.

We discuss three main consistency models that can be implemented by transactional memory systems: *2-phase locking*, *linearizability* and *conflict-serializability*. We demonstrate that 2-phase locking is more strict than linearizability and conflict-serializability, allowing fewer schedules to be valid.

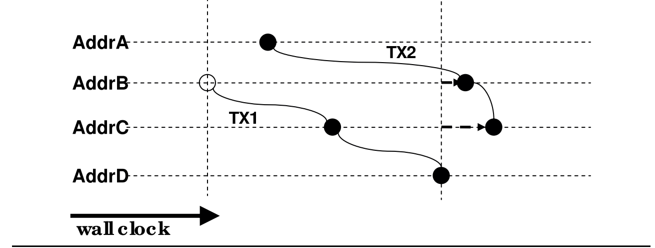


Figure 1. A schedule valid under 2PL.

2.2.1 2-Phase Locking (2PL)

2PL does not allow conflicting accesses while a transaction is running. If such an access occurs, either one of the conflicting transactions is aborted, or the conflicting access is delayed. 2PL is illustrated in Figure 1, which shows the schedule of two transactions TX1 and TX2 that make read and write accesses to addresses A, B, C, and D. The read actions are represented by empty circles, whereas write actions are represented by solid circles. In the schedule shown, TX1 and TX2 perform conflicting accesses on addresses B and C. Thus, either one of the transactions must be aborted (obstruction-free software TM systems), or TX2 must wait until TX1 completes (blocking software TM systems) as indicated by the dashed arrow in Figure 1. Conversely, TX2 can perform its access to address A without aborting or stalling because this access does not conflict with TX1.

Note that in this paper, we use the term *2PL* to refer to a consistency model rather than an actual implementation. That is, a TM system can *imitate* a locking-based implementation (i.e. admit the same schedules) even if it is not lock-based. For instance, the software or hardware TM systems that implement optimistic concurrency models (such as TCC [5] and TL2 [6]) allow active transactions to commit even if they show their *intentions* to write to the same locations during execution. However, these systems in fact implement the 2PL consistency model by delaying the actual writes until commit time, just like in the example shown in Figure 1.

2.2.2 Linearizability

A schedule S is said to be linearizable if:

1. It is view-equivalent to some legal serial schedule S_{ser}, and
2. The transaction partial order induced by S is a subset of the operation partial order induced by S_{ser}.

The first rule states that if a schedule is linearizable, it should be possible to obtain a serial legal schedule S_{ser} by just reordering the actions of transactions in such a way that the final outcome of execution would remain the same. The second rules imposes that the order of transactions must remain the same in S_{ser}. That is, if a transaction TX1 completes before another transaction TX2 starts executing, all the actions of TX1 must come before all the actions of TX2 in S_{ser}. In fact, we can say that under linearizability, the execution is equivalent to some execution such that each transaction happens instantaneously at some point, called *linearization point* between the start and commit. Finding a linearization point for all the transactions is called a *linearization*.

The schedule in Figure 2 is invalid under 2PL because TX2 makes conflicting accesses on addresses B and C before TX1 completes its execution. The schedule, on the other hand, is linearizable because TX1 → TX2 → TX3 (i.e. all the actions of TX1 precedes

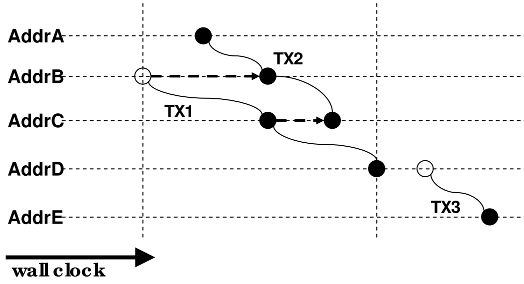


Figure 2. A linearizable schedule invalid under 2PL.

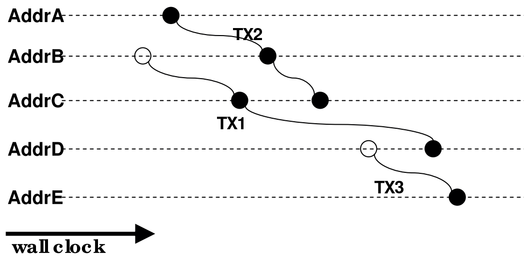


Figure 3. A serializable but non-linearizable schedule.

all the actions of TX2 and all the actions of TX2 precedes all the actions of TX3) is a valid linearization. This linearization produces the same outcome as the concurrent execution (the same values are read and written) and respects the partial order of transactions (TX3 comes after TX1 and TX2).

2.2.3 Conflict-Serializability

A schedule is said to be conflict-serializable (or simply serializable) if it is conflict-equivalent to some legal serial schedule. Thus, serializability, unlike linearizability, does not require a partial order of transactions to be preserved. The schedule in Figure 3 is serializable because TX3 → TX1 → TX2 is a valid ordering of transactions with respect to the order of the conflicting accesses. However, this schedule is not linearizable because linearizability imposes the additional ordering of TX2→TX3 based on the order of the start and commit times of transactions. This requirement contradicts the previous ordering of transactions which is based on the order of the conflicting accesses.

Note that conflict-serializability is based on finding conflict-equivalent serial schedules, and hence it is different from other models such as sequential consistency, linearizability, view serializability and strict serializability which are based on view-equivalence of schedules. Thus, just like 2PL and linearizability, conflict-serializability also presents an intiutive model to the programmer, where transactions observe the final committed values of shared data. Yet, conflict-serializability is more relaxed than 2PL and linearizability because it does not impose any order on transactions and, therefore it allows more concurrency.

2.3 SON-Method for Serializability

In our previous work [2], we proposed a method for efficiently implementing conflict-serializability in a software transactional memory system. We showed that the use of serializability is beneficial to performance, particularly for applications that have a large degree of data sharing, and thus are difficult to execute efficiently with 2PL.

Our method attempts to incrementally construct a conflict-equivalent sequential 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. During execution, 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 must be the same, as described above. If a unique SON can be determined for each transaction, then a conflict-equivalent serial schedule exists. Further, we can conclude that all read/write/commit actions of transactions atomically happen in the order of their transaction's SONs.

The SONs of transactions are determined using the following two basic rules:

1. If a transaction TX1 accesses (reads or writes) an address that has already been committed by another transaction TX2, then TX1's SON must be be higher than that of TX2. This is because the access of TX1 happen later than the access of TX2.
2. If a transaction TX1 reads an address that is later committed by another transaction TX2, then TX1's SON must be lower than that of that of TX2. This is because TX1's read action happens earlier than the commit action of TX2.

The first rule imposes a lower bound on the SON of a transaction. Similarly, the second rule imposes an upper bound on the SON of a transaction. Thus, we associate with each transaction a pair of integer values that reflect the lower and upper bounds on its SON. The lower bound of a transaction is initialized to 0, and the first rule is repeatedly used to increase it. The upper bound is initialized to ∞ and the second rule is used to lower it. If at any moment during execution, the lower bound on the SON for 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 its commit phase during which it examines its SON range and determines a specific SON. If the upper bound of the range is not ∞, then it reflects the SON of some conflicting transaction. Therefore, 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.

2.4 Hardware TM Systems

Previous studies on TM have shown that minimal hardware support can provide fast transactions [1, 7] in spite of limitations of available resources. Cache coherence protocols provide elegant and efficient ways for detecting conflicts, with minimal extensions over standard protocols. In addition, hardware buffers, broadcasting, hardware tokens, transactional status bits for cache lines are all simple hardware techniques that can be used to provide isolation of stores and fast atomic commits for transactions. Below we discuss different design choices that can be implemented in a hardware TM system.

1. **Eager/Lazy Conflict Detection (CD)**. With eager conflict detection, conflicts are immediately detected as the accesses happen. With lazy conflict detection, on the other hand, conflicts

[2] The selection of integer numbers as SONs imposes the additional constraint that lower bound cannot be upper bound minus one.

are detected when transactions commit and broadcast their updates to other transactions.

2. **Eager/Lazy Version Management (VM)**. With eager version management, active transactions perform their updates in place (in cache or memory); each keeps a log of old-values. With lazy version management, old values are kept in place and updates are buffered in a private per-processor transactional buffer until commit time. Lazy version management provides fast aborts, because the old values are kept in place, whereas the eager version management provides fast commits because the new values are kept in place.
3. **Conflict Management (CM)**. When a conflict happens, conflict management algorithm dictates what type of action must be taken. Three types of actions can be taken depending on which transaction wins the conflict: *committer wins*, *requester wins* and *requester stalls*. With requester stalls, updates that conflict with previous accesses are stalled until the accessor transaction commits. A conflict causes an abort only if a stall potentially cause a deadlock.

Depending on the design choices they implement, we can classify hardware TM systems into three main types [4]:

1. **Lazy CD/Lazy VM/ Committer Wins**. These systems are referred to as *LL systems*. TCC [5] and Bulk [8] are the main hardware systems of this type. To perform atomic writes, LL systems acquire commit tokens [9], and to ensure forward progress, they follow the committer wins policy.
2. **Eager CD/Lazy VM/Requester Wins**. These systems are referred to as *EL systems*. LTM [10] is an example TM system of this type.
3. **Eager CD/Eager VM/Requester Stalls**. These systems are referred to as *EE systems*. LogTM implementations [11, 7] are the main representatives for this type of hardware support.

The well-known hardware TM systems proposed in the literature are based on the 2PL consistency protocol. That is, a transaction can commit if and only if it does not conflict with another transaction from start to commit. Any conflict causes one of the conflicting transactions to abort or to be delayed.

3. Hardware-Based Implementation of Serializability

In this section, we discuss how serializable transactions can be implemented using our SON-based technique by taking advantage of a base hardware support for transactional memory. We start with the description of this base support. Then, we present the additional support required to implement our SON-based technique for serializability on top of this basic hardware support. Specifically, we discuss the additional transaction metadata that needs to be stored, and the operations that need to be performed for each transactional load, transactional store, conflict and transaction commit. We discuss both a software/hardware hybrid implementation and in generic terms a full hardware implementation. The hybrid implementation assumes only the minimal TM hardware support of the base system; serializability is implemented as a software layer on top of the hardware. The full hardware implementation incorporates serializability support in hardware by extending the base hardware TM support and assumes minimal compiler support.

3.1 Base Hardware TM Support

The base hardware system we consider is an LL system with lazy conflict detection/lazy version management [4]. This choice is mandated by the fact that SONs are selected at commit time. More specifically, the second rule of our SON-based method states that if a transaction writes to an address already accessed (read or written) by another transaction, the upper bound of the accessor transaction must be updated with the SON of the writer transaction. Since SONs are determined at commit time, our technique requires that conflicts are detected when the writer transaction commits. However, when the write access actually happens, the writer transaction is not yet assigned an SON (let alone know if it will successfully commit). Thus, the hardware support for our SON-based technique must use lazy conflict detection that detects the conflicts when transactions commit their writes. This allows the SON of the committing transaction to be communicated with other conflicting transactions.

3.2 Serializing Transactions

We believe that serializability can be implemented as the consistency model using our SON-based technique, on top of our base system. Below, we discuss what type of additional support (either in software or in hardware) is necessary for this purpose.

- **Transaction Metadata**. In addition to the traditional read and write sets kept for detecting conflicts, an SON lower bound and an SON upper bound must be stored for each transaction.
- **Transactional Load**. When a load instruction is issued, the transaction must (1) update its lower bound with the write-number associated with the address, (2) verify that the transaction itself is still valid by comparing its lower and upper bounds. These operations are in addition to the operations performed by a typical TM system, such as adding the loaded address to the read-set.
- **Transactional Store**. When a store instruction is issued, our SON-based technique does not require any additional operation. Since the store value is buffered and the address can be written to by other transactions several times until the transaction commits, it is more efficient to perform the serializability check (updating lower bound and comparing it with upper bound) at commit time.
- **SON-Tables**. We need to store: (1) For each address written to by any transaction, the SON of the transaction that wrote to the address, which we refer to as the *write-number*, (2) For each address read by any transaction, the SON of the last transaction that read the address and committed successfully, which we refer to as the *read-number*.
- **Conflicts**. In traditional transactional memory implementations, conflicts cause transactions to abort. In our SON-based technique, a conflict triggers a serializability check. Since we are assuming a LL system, a conflict happens between a transaction that is trying to commit a value to an address, and a transaction that already accessed the same address. The committer transaction cannot abort, and thus the reader transaction should update its upper bound with the SON of the committer transaction and verify that it is still valid. For this purpose, the conflict triggers a validation, during which the reader transaction (1) iterates over its read set to find which address was updated since the last validation, (2) updates its *own* upper bound with the write-number associated with this address.
- **Transaction Commit**. When a transaction commits, it must validate its writes. For this purpose, it (1) iterates over its write set and updates its lower bound with the write-number associated with each address it wrote to, (2) iterates over its read set and updates its lower bound with the read-number associated with each address it read, (3) verifies that it is still valid. After the validation, the transaction selects an SON and commits by (1) updating write-numbers for each address in its write-

```
Transaction* tx;
START_HW_TX();
tx = START_SW_TX();
for(Node* n = head->next(); n != NULL; n = n->next()) {
  if(tx->openForRead(n) == false) ABORT_HW_TX();
  if(n->key() == key) break;
}
If(tx->COMMIT_SW_TX() == false) ABORT_HW_TX();
COMMIT_HW_TX();
```

Figure 4. Lookup method under the hybrid programming model.

set, (2) updating read-numbers for each address in its read-set. Then, the transaction is ready to perform the traditional hardware commit operation storing the values in memory and informing other transactions of conflicts.

3.3 Hybrid Implementation

In this section, we discuss how the above-mentioned additional support for our SON-based technique can be implemented as a software layer on top an LL system with only 2PL-based hardware support. For this purpose, we need to (1) implement library calls for transactional load, store and commit actions, (2) store transactional metadata in software, (3) maintain write-numbers and read-numbers in software and (4) implement software traps for conflict handling. We discuss how these are done below.

Library calls are used to perform serializability checks, before transactional load, store and during commit actions. Because inserting calls before each load and store instruction is inefficient and impractical, we choose to use object-level conflict detection in the software layer, even though conflicts are still detected at the cache line level by the hardware. That is, before accessing a field of an object, software routines are called to *open* the object in either read or write mode at which point serializability checks are performed. Thus, shared objects of target applications inherit from a *transactional object* class which implements these open methods and other transactional data members that will be mentioned later. Figure 4 shows how the library calls are used for the lookup method of a linked list class.

The transactional metadata consists of an SON lower bound, an SON upper bound, a read-set and a write-set for each transaction and stored in a *Transaction* class instance. The read-numbers and write-numbers are stored as data members of transactional objects. These numbers are read to perform serializability checks inside open and commit library calls, and they are updated during commit with the SON of the committing transaction. Since transactional objects are shared by all transactions, accesses to these data members are subject to race conditions. To avoid these race conditions, readings of the read-numbers and the write-numbers are performed as non-transactional operations. However, updates are performed as transactional operations. This ensures that updates of read-numbers and write-numbers are atomic and are discarded if the transaction aborts.

Software traps are used to switch to software routines to perform serializability validation when conflicts occur. Conflicts are triggered during the execution of an active transaction, if the transaction has already accessed an address which is currently being committed by another transaction. At this point, the active transaction traps to a software trap handler which calls a validation method for this transaction. Before the trap is serviced, the program context (program counter, registers, etc.) are saved to allow the transaction to continue execution after the trap. The validation method iterates over the read-set of the active transaction and tries to find the objects that were committed since they were opened in read mode by the transaction. These are the objects whose write-numbers have changed since they were opened. If such an object is found, the active transaction updates its upper bound with the write-number of the object and verifies that the upper bound is still higher than the lower bound. If the transaction is still valid, it continues execution by re-loading its saved context. If the transaction is invalid, it executes an abort instruction which causes the hardware transaction to abort and restart.

Note that, in order to avoid inconsistent state, the validation method must update its upper bound with the write-number of the object associated with the conflicting address that triggered the validation. That is, the write-number of this object must be visible to the active transaction when it performs the validation. Because write-numbers are updated with transactional actions, these updates are buffered inside hardware transactional buffers and become visible to other transactions only when transactions commit. Thus, we must make sure that a committer transaction commits the write-numbers of transactional objects first before it commits its updates which may cause conflicts. We can simply ensure this by making transactions to commit their updates in a LIFO (last in first out) order. This is because software commits are invoked just before hardware commits and hence write-numbers are always the last entries in transactional buffers.

Further, while a conflict is being handled, other conflicts can still be triggered by committing transactions. These conflicts are buffered as they occur, and the active transaction services them one by one.

Finally, we should note that hardware TM systems (such as Rock [3]) may cause transactions to abort not only due to conflicts, but also due to the resource limitations such as overflow of speculative caches, buffer overflows, etc. These cases do not cause software traps and triggers immediate aborts of transactions. If the hardware resources are exceeded during the software validation (for instance if a speculatively loaded address is kicked out of the cache due to a capacity miss), the abort trap is recorded and serviced when the transaction completes its software validation.

Figure 5 depicts the transactional operations performed at the software layer.

3.4 Full-Hardware Implementation

In this section, we describe how $\mathcal{CS}$ can be implemented fully in hardware with our SON-based method. For this purpose, we need to implement the additional features discussed in Section 3.2 to the standard LL system we assume, effectively replacing the support for 2PL with support for $\mathcal{CS}$. Below, we describe in generic terms what such implementation looks like.

For each transaction, we need to store an SON lower bound and an SON upper bound. Two general purpose registers, *lb* and *ub* can simply be reserved during the execution of transactions for this purpose. Simple comparison instructions on these registers can be used to verify transactions (i.e. $ub > lb$). A compiler can efficiently place these comparison instructions inside transactional code (for instance every few instructions, at the end of every loop iteration, etc.) in such a way to prevent inconsistency problems (arithmetic faults, memory access violations, infinite loops, etc.) similar to validation checks in software TM systems [12, 13, 14].

For each cache-line, we need to store a write-number and a read-number in order to update the transaction SON lower and upper bounds at transactional load and commit operations. Two techniques can be used for this: augmented cache lines and virtual tables. With augmented cache lines, the write numbers and the read numbers are stored in the cache lines at a particular offset. The compiler can easily use padding techniques to allocate space in each cache line for this purpose. However, doing this for the

```
bool Transaction::openForRead(txObject* o) {
 addtoReadList(o, o->getWriteTime());
  return updateLowerBound(o->getWriteTime());
  }
void Transaction::openForWrite(txObject* o) {
  addtoWriteList(o);
}
bool Transaction::validate() {
  for(int i=0; i <readListSize; i++) {
    unsigned long writeTime = readList[i]->getWriteTime();
    // obj has been committed since tx read it
    if(writeTime != readList[i]->getTime(i))
      if(updateUpperBound(writeTime) == false)
        return false;
  }
  return true;
}
bool Transaction::commit() {
  for(int i=0; i < writeListSize; i++) {
    if(updateLowerBound(writeList[i]->getWriteTime()) == false)
      return false;
    if(updateLowerBound(writeList[i]->getReadTime()) == false)
     return false;
  }
  unsigned long SON = assignSON();
  for(int i=0; i<writeListSize; i++) writeList[i]->setWriteTime(SON);
  for(int i=0; i<readListSize; i++) readList[i]->setReadTime(SON);
  return true;
}
```

Figure 5. Transactional operations under the hybrid model.

entire address space accessed by the program can significantly increase the memory footprint. Hence, the user can specify the structures/addresses that will be accessed inside transactions, so that compiler can pad only these addresses at memory allocation time. This is similar to the programming model of most software TM systems [15, 13].

The virtual tables technique, on the other hand, is based on allocating a specific virtual address space to store tables of write numbers and read numbers, indexed by the appropriate lower bits of memory addresses. Although this technique does not require the analysis of shared address space, because the tables will be of a specific size, aliasing of addresses can cause unnecessary aborts.

We elect to use augmented cache lines to store the write-numbers to avoid aliasing. However, this technique is not suitable for read-numbers. This is because for every cache line speculatively loaded, transactions need to update *only* the read numbers in the cache line, not the data. This is hard to do because the actual data in the cache line may have been changed in memory since transaction loaded it. Therefore, we elect to use virtual tables for storing read-numbers.

Transactional load instruction will update the lower bound of a transaction with the write-number of the cache-line. As discussed above, the lower bound is stored in a special register *lb* and the write-number is either stored at a particular offset in the cache line or a specific address in the virtual memory.

```
txload (addr):
  if (lb < [addr + offset]) lb = [addr + offset]
  insert (readBuffer,addr)

txstore (addr):
  insert (writeBuffer,addr)

conflict (remoteSON):
  if (remoteSON < ub) ub = remoteSON

txcommit:
   acquireToken;
   for each w in writeBuffer                      \
     if (lb < [w +offset]) lb = [w + offset]       |  serializabilty checks
     if (lb < readTime[w]) lb = readTime[w]        |
   if(ub <= lb) { releaseToken, abort }           /
   if(ub == 0) ub = lb + numProc;                 \  selecting SON
   else ub = ub – 1;                              /
   for each w in writeBuffer                      \
     [w + offset] = ub;                            |
     broadcast(w, ub)                              |  write-number and
   for each r in readBuffer                        |  read-number updates
     readTime[r] = ub                             /
  releaseToken
```

Figure 6. Transactional operations in hardware.

Other than standard transactional operations (storing the data in a store-buffer, setting cache flags, etc.), a transactional store instruction does not perform any additional transaction operation in our SON-based technique. The validity of stored addresses will be verified at commit time.

A conflict will occur when a transaction is committing a value to an address previously loaded by another transaction. In this case, the upper bound of the reader transaction must be updated with the SON of the committer transaction. This can simply be achieved by broadcasting the SON of the committer transaction when the value is being committed to the memory. When the reader transaction detects the conflict, it updates its own upper bound.

Finally, before a transaction commits its modifications, its updates must be validated. For this purpose, the processor iterates over the write buffer and updates its lower bound with the write-number and read-number of each stored address. If transaction becomes invalid, the commit fails. Otherwise, the transaction is assigned an SON and proceeds to committing read and write numbers. Because the SON lower bound and the SON upper bound will not be needed after the transaction commits, the *lb* and *ub* registers can be used to store the SON of the transaction during commit. Committing values is performed by iterating over the write set and writing the values to the main memory together with the write-numbers. The read-numbers also need to be updated; the transaction iterates over the read-set and stores the read-numbers in the virtual table.

Figure 6 describes the transactional hardware operations for serializability.

4. Experimental Evaluation

In this section, we describe our experimental evaluation of the two implementations of our SON-based technique. We use the breakdown of conflicts and the breakdown of transaction execution cycles as evaluation metrics.

4.1 The Experimental Framework

We build our SON-based algorithm on top of a standard LL hardware TM implementation discussed in [4]. This system provides base support for TM as described in Section 3.1. It is built using Simics [16] full-system simulation infrastructure. The Wisconsin GEMS toolset [17] 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 onto the memory model. The simulated target system runs Solaris 10 to provide OS support for applications.

For our simulations, we model a 8-core CMP version of this implementation. Each core is in-order and single-issue sparc processor. Each processor has 32 KB private writeback L1 I & D caches. The 8 MB L2 cache is shared and consists of 32 banks interleaved by block address. Cores and cache banks are connected with a packet-switched tiled interconnect consisting of 2 clusters with 4 cores in each. Four on-chip memory controllers connect to standard DRAM banks. Cache coherence is implemented with the MESI protocol and maintained with an on-chip directory at L2 cache backs.

4.2 Benchmarks

In order to assess the impact of serializability on performance, we used two standard microbenchmark applications commonly used to evaluate TM systems:

- Linked-List (List). This is a straight-forward implementation of an ordered linked-list, where each node contains a single key and the nodes are singly-linked to each other. Keys range between 0 and 1023. The list is randomly initialized to contain 1024 nodes and each processor performs 512 operations consisting of 1:1:1 mix of insert, delete and lookup operations. Since there is extensive data sharing between transactions in this benchmark, it allows us to evaluate the impact of serializability when abort rates are high.
- Red-black Tree (RBT). This is the RBT implementation used in [2]. Each node in the tree contains a single key and transactions perform a 1:1:1 mix of insert, delete and lookup operations. The randomly generated keys range between 0 and 16383, which limits the depth of the tree to 14 when the tree is fully balanced. Keys range between 0 and 1023. The list is randomly initialized to contain 16384 nodes and each processor performs 2048 operations, consisting of 1:1:1 mix of insert, delete and lookup operations. The tree structure limits the data sharing among transactions in this benchmark. Therefore, this benchmark is useful to evaluate serializability under low-abort rates.

4.3 Evaluated Systems

We evaluate three systems.

- LL. In order to evaluate a standard transactional memory system which is based on the 2PL consistency model, we use the LL hardware transactional memory system discussed in [4]. Analyzing hardware resource constraints is outside the scope of this paper, and thus we assume an infinite write buffer in order to eliminate transactional buffer overflows.
- Hybrid Implementation of Serializability (HybSON). This is the hybrid implementation of serializability described in Section 3.3. The underlying TM support is provided by the same LL system mentioned above.
- Ideal Hardware Implementation of Serializability (IHwSON). In order to evaluate the maximum performance gain that can be achieved using our SON-based technique, we implemented a hardware system that provides ideal support for serializability. This implementation is ideal in the sense that it does not incur any additional overheads due to the SON-based technique. That is, accessing SON-tables, updating SON lower and upper bounds, serializability checks are implemented as no-overhead operations on top of the standard LL hardware support described above. There are two reasons why we pursue this ideal implementation. The first is that at this stage our goal is to determine the limits on performance gains in hardware-based TM systems. The second reason is that we expect the overheads to be minimal in the full implementation because the required extensions are simple.

  The main advantage of our SON-based technique over LL is lower abort rates due to the more relaxed serializability consistency model. Hence, by comparing the performance of this ideal hardware implementation with the performance of the standard hardware support, we can evaluate the upper bound on the performance gain we can obtain by using our technique.

4.4 Experimental Results

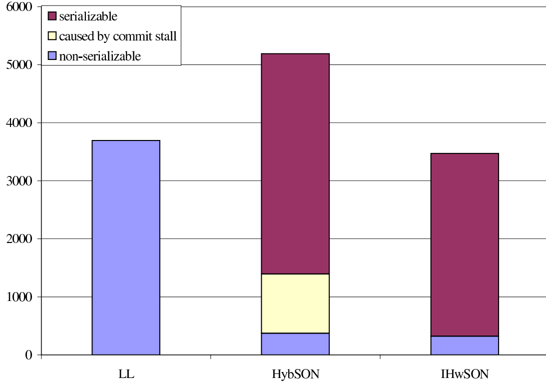
We simulated an 8 processor system using the LL simulator described above, and compared the evaluated systems in terms of the number of conflicts, total execution cycles and cycles wasted due to aborts.

4.4.1 Conflicts

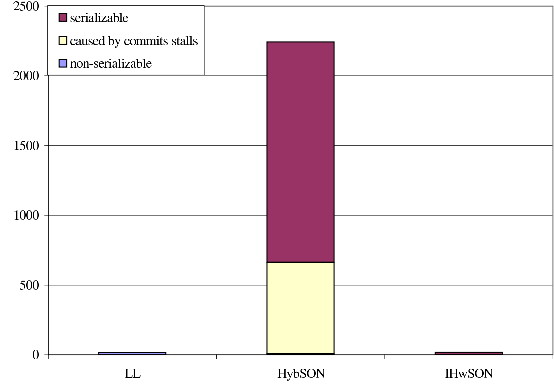
Figure 7 shows the breakdown of conflicts that occur during the execution of benchmarks. LL does not support serializability, and thus all the conflicts are non-serializable and cause aborts. With HybSON and IHwSON, serializable conflicts do not cause conflicts; transactions continue after validation. With HybSON, transactions perform serializability checks and update SON tables. This prolongs the time it takes transactions to commit. Because transactions acquire tokens to serialize commits, they stall until the commit token is available, during which accesses of active transactions may conflict with the accesses of this stalled transaction. If such a conflict occurs, the stalled transaction immediately aborts without trapping to software for serializability validation. These conflicts are labeled as the "caused by commit stall" bar in the figure.

For List, the number of non-serializable conflicts is significantly lower in HybSON and in IHwSON compared to LL. More specifically, we see that 90.7% (3152 out of 3473) of conflicts in IHwSON are serializable and do not require aborts. The figure also shows that the total number of conflicts remains almost the same for LL and IHwSON systems. This matches our expectations because the consistency model implemented does not have any impact on the accesses of transactions and how the accesses conflict. We notice that HybSON incurs more conflicts in total. This is due to the conflicts caused by commit stalls as described above. If we exclude these conflicts, the total number of serializable and non-serializable conflicts are the same in HybSON and IHwSON. This again shows that implementing a different consistency model does not affect the access patterns of applications and hence the number of conflicts.

For RBT, both LL and IHwSON incur few conflicts due to the tree structure of the application. The number of non-serializable conflicts is almost the same in HybSON and IHwSON. However, HybSON incurs significantly more non-serializable conflicts. This



(a) Linked list.



(b) Red-black tree.

Figure 7. Breakdown of conflicts.

Table 1. Execution Cycles for Linked list.

	Wasted Cycles	Total Cycles
LL	3, 576, 285	13, 840, 741
HybSON	4, 235, 209	64, 843, 198
IHwSON	404, 512	11, 028, 933

Table 2. Execution Cycles for Red-black tree.

	Wasted Cycles	Total Cycles
LL	38, 582	9, 376, 218
HybSON	97, 200	18, 934, 679
IHwSON	67	9, 675, 622

is due to the fact that, as it will be discussed below, transactions take longer to execute in `HybSON`. Thus, while transactions are too short to conflict in `LL` or in `IHwSON`, they are more susceptible to conflicts in `HybSON` due to longer execution. We also see that `HybSON` suffers from a number of conflicts caused by commit stalls.

4.4.2 Cycles

Table 1 and Table 2 show the total execution cycles it takes to run each benchmark and the execution cycles for each processor wasted due to aborted transactions. For `List`, we see that `IHwSON` take the fewest cycles to execute. It is 20.3% faster compared to `LL`. The difference in the number of execution cycles between `LL` and `IHwSON` (2,811,808) is almost equal to the difference in the number of wasted cycles due to the aborts (3,171,765). Thus, we can conclude that `IHwSON` is faster than `LL` mostly because the number of wasted cycles due to aborts is lower. Further, for `HybSON`, the execution is 4.6 times longer compared to `LL`. This is due to the overheads of software layer that perform extra bookkeeping and checks for serializability. These overheads also increase wasted cycles as shown in the figure.

For `RBT`, again `IHwSON` takes the fewest cycles to execute, and `HybSON` takes the most due to the software overheads. Because the abort rates are very low and transactions are short, the impact of wasted cycles on performance is very small.

Figure 8 shows the breakdown of total execution cycles. We notice similar patterns for both benchmarks. First, for `HybSON`, we see that 26% of execution time is spent in software traps for serializability checks on average. Further, 10% of execution time is spent for software validation, another 10% is spent for commit operations, and 25% of execution time is spent on stalling for commit token. These represent the overheads of having a software layer for implementing serializability. Second, `IHwSON` transactions spend less time on transactional operations on average, even though the time spent for non-transactional operations is the same in `LL` and `IHwSON`. This is due to the impact of fewer aborts and hence, fewer wasted cycles. Finally, for `LL`, the execution time mostly consists of time spent for transactional actions and non-transactional instructions. The impact of commit time and aborting transactions is insignificant.

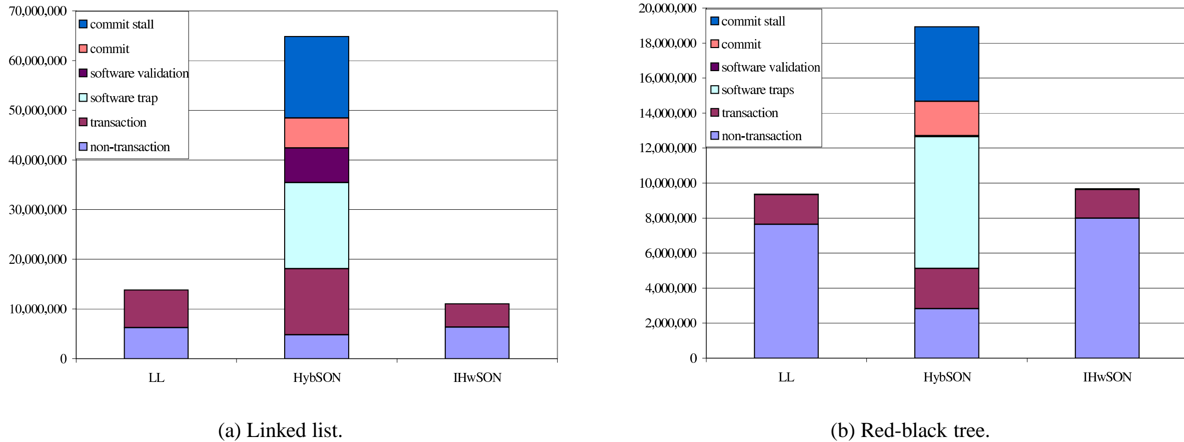
We have been limited by the time to run the simulations. This in turn, limited the length of transactions in our benchmark applications in comparison to what we used in the past for our software-based implementation of $\mathcal{CS}$. We expect that with longer running transactions that the benefit of the full hardware implementation will only improve.

5. Related Work

In our previous work [2], we described how conflict-serializability can be implemented using our SON-based technique in a software TM system. We demonstrated that such a relaxed consistency model can improve performance by lowering the abort rates, especially for applications with long running transactions and high levels of data sharing. Conflict-serializability is a well-known concept in the database domain [18]. Thus, our contribution in this regard was to successfully employ and implement this model in the domain of TM systems.

There has been several other approaches to relaxing consistency models in TM systems. Calstrom et al. [19] assert that long-running transactions are important for the ease of parallel programming and recognize that the use of 2PL limits concurrency. They propose transactional collection classes to reduce the number of conflicts and demonstrate its use for some example data-structures. However, their approach requires knowledge of the semantics of data structures and the dependencies that exist.

Riegel et al. [20, 21] investigate the use of snapshot isolation, which allows transactions to access old versions of shared data when there exist conflicts over the new versions. Snapshot isola-



(a) Linked list.

(b) Red-black tree.

Figure 8. Breakdown of transaction execution.

tion does not provide serializability nor linearizability when old versions are used. It requires that programmer should carefully analyze the accesses and transform some read accesses to write accesses for correctness. If linearizability is required, snapshot isolation again resorts to 2PL to ensure correctness.

Several software/hardware hybrid implementations of TM systems have been proposed in the literature. Most noticeably, Damron et al. [22] proposed the Hybrid Transactional Memory system that switches between hardware and software transactions depending on the availability of hardware resources. Further, Lev et al. proposed the Split Hardware Transactions that implement a software layer on top of a standard hardware support for TM. The functionality of the software layer was to provide atomicity for parent transactions that are split into multiple child hardware transactions. Both of these hybrid TM implementations are based on the 2PL consistency model, whereas in our work we use the software layer to relax the consistency model used by the hardware implementation.

6. Conclusions

In this paper, we discussed how conflict-serializability consistency model can be supported in a base TM hardware system with lazy conflict detection and lazy version management. We discussed the type of additional features that are necessary to implement conflict-serializability in any hardware-based system. We further described both a hybrid system where serializability is implemented as a software layer on top of a base hardware system, as well as a system that extends the base system to support conflict-serializability fully in hardware.

Using a simulator and two representative benchmarks, we evaluated both the hybrid system and an ideal version of the full hardware system that incurs no overheads due to serializability. This ideal system gives us an upper bound on the performance gains that can be achieved using serializability before we implement the full hardware system. Our evaluation shows lower abort rates in both cases. However, the hybrid system is 2-5 times slower compared to the base hardware due to the overheads of bookkeeping and consistency checks in software. The ideal hardware system, on the other hand, is faster than the base system because the cost of aborts is lower. This difference in performance can be significant (23.2% for linked list) when the cost of aborts is high in the base hardware system, such as in applications with long running transactions and high degree of conflicts.

Based on our results, we conclude that the hybrid solution is not viable for typical benchmarks, and should only be considered for applications that suffer from excessive aborts yet with small amount of bookkeeping. The ideal hardware system, on the other hand, validates our expectations that reducing the abort rates positively affect overall performance. We believe that an efficient hardware implementation of serializability can take advantage of the SS consistency model to improve performance.

References

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

[2] U. Aydonat and T. S. Abdelrahman, “Serializability of transactions in software transactional memory,” in *TRANSACT ’08: Workshop on Transactional Computing*, 2008.

[3] M. Moir, K. Moore, and D. Nussbaum, “The adaptive transactional memory test platform: a tool for experimenting with transactional code for rock (poster),” in *SPAA ’08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pp. 362–362, 2008.

[4] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, “Performance pathologies in hardware transactional memory,” *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 81–91, 2007.

[5] 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 ISCA*, p. 102, 2004.

[6] D. Dice, O. Shalev, and N. Shavit, “Transactional locking ii,” in *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[7] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, “Logtm-se: Decoupling hardware transactional memory from caches,” in *HPCA ’07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 261–272, 2007.

[8] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, “Bulk disambiguation of speculative threads in multiprocessors,” in *ISCA ’06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pp. 227–238, 2006.

[9] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun, “Architectural semantics for practical transactional memory,” in *ISCA ’06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pp. 53–65, 2006.

[10] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, “Unbounded transactional memory,” in *HPCA ’05: Proceed-*

ings of the 11th International Symposium on High-Performance Computer Architecture, pp. 316–327, 2005.

[11] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, “Logtm: Log-based transactional memory,” in *Proceesings of the 12th Annual International Symposium on High Performance Computer Architecture (HPCA-12)*, 2006.

[12] T. Harris and K. Fraser, “Language support for lightweight transactions,” in *Proc. of OOPSLA*, pp. 388–402, 2003.

[13] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer, “Software transactional memory for dynamic-sized data structures,” in *Proc. of PODC*, pp. 92–101, 2003.

[14] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, “Mcrt-stm: a high performance software transactional memory system for a multi-core runtime,” in *Proc. of PPoPP*, pp. 187–197, 2006.

[15] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott, “Lowering the overhead of software transactional memory,” Tech. Rep. TR 893, Computer Science Department, University of Rochester, 2006.

[16] M. P.S., C. M., E. J., F. D., H. G., H. J., L. F. M. A., and W. B., “Simics: A full system simulation platform,” Feb 2002.

[17] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, “Multifacet’s general execution-driven multiprocessor simulator (gems) toolset,” *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[18] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[19] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun, “Transactional collection classes,” in *In Prof. of PPoPP*, pp. 56–67, 2007.

[20] T. Riegel, C. Fetzer, and P. Felber, “Snapshot isolation for software transactional memory,” in *TRANSACT, http://wwwse.inf.tu-dresden.de/papers/preprint-riegel2006sistm.pdf*, 2006.

[21] H. Attiya, R. Guerraoui, and E. Ruppert, “Partial snapshot objects,” in *SPAA ’08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pp. 336–343, 2008.

[22] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, “Hybrid transactional memory,” in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pp. 336–346, 2006.