Software Transactional Memory Validation – Time and Space Considerations

Adam Welc Bratin Saha

Programming Systems Lab
Intel Corporation
Santa Clara, CA 95054
{adam.welc,bratin.saha}@intel.com

Abstract

With single thread performance hitting the power wall, hardware architects have turned to chip-level multiprocessing to increase processor performance. As a result, issues related to the construction of scalable and reliable multi-threaded applications have become increasingly important. One of the most pressing problems in concurrent programming has been synchronizing accesses to shared data among multiple concurrent threads.

Traditionally, accesses to shared memory have been synchronized using lock-based techniques resulting in scalability, composability and safety problems. Recently, transactional memory has been shown to eliminate many problems associated with lock-based synchronization, and transactional constructs have been added to languages to facilitate programming with transactions. Therefore, providing an efficient software transactional memory (STM) implementation has been an important area of research. One of the largest overheads in an STM implementation is incurred in the validation procedure (that is, in ensuring correctness of transactional read operations).

This paper presents novel solutions to reduce the validation overhead in an STM. We first present a validation algorithm that is linear in the in the number of read operations executed by a transaction, and yet does not add any overhead to transactional reads and writes. We then present an algorithm that uses bitmaps to encode information about transactional operations and further reduces both the time and space overheads related to validation. We evaluate the effectiveness of both algorithms in the context of a state-of-the-art STM implementation.

1. Introduction

With single thread performance hitting the power wall, hardware architects have turned to chip-level multiprocessing (CMP) to increase processor performance. All major processor vendors are aggressively promoting CMPs in the mainstream computing market. In a CMP environment, applications have to be concurrent to exploit the computing power of the hardware platform. As a result, issues related to the construction of scalable and reliable multi-threaded applications have become increasingly important. One of the most pressing problems in concurrent programming has been the mediation of accesses to shared memory by multiple concurrent threads of execution.

Today, programmers use lock-based synchronization to manage concurrent accesses to shared memory. However, lock-based synchronization leads to a number of software engineering problems as well as scalability bottlenecks arising from over-synchronization, which is often a result of programmers attempting to ensure correctness. Transactional memory [10, 5, 3, 4, 6] avoids many problems associated with lock-based synchronization by eliminating deadlocks, providing safe composition, and enabling application scalability through optimistic concurrency. Accordingly, there has been a lot of recent interest in adding transactional constructs to programming languages, and in providing high performance implementations of transactional memory.

Transactional memory (TM) can be implemented both as a hardware (HTM) [7, 9, 8], or as a software (STM) [11, 10, 5, 4] system. For use as a programming construct, a TM system must support transactions of unbounded size and duration, and allow transactions to be integrated with a language environment. These requirements are currently fulfilled only by software TM systems, and it is reasonable to assume that any virtualized TM system would include an STM implementation at its core. Thus, a high performance STM is crucial for deploying transactional memory as a programming mechanism.

An STM implementation instruments all loads and stores inside a transactional code block to generate the meta-data that it needs to guarantee atomicity and isolation [2] of the transactional code block. In order to ensure good scalability, an STM implementation must also allow optimistic read concurrency [10, 5, 12], wherein the run-time system optimistically executes a transaction, and then validates all the read operations before committing its results. Validation constitutes one of the major costs of an STM implementation [10]. In this paper, we present a number of novel mechanisms to reduce the STM validation cost, and thereby make the STM implementation more efficient. We also evaluate the effectiveness of the solutions on a number of standard transactional workloads.

This paper makes two novel contributions:

- We present and evaluate a validation algorithm that is linear in the number of read operations executed by a transaction, does not add overhead to STM read and write operations, and does not suffer from false conflicts. Other solutions we are aware of are either of higher complexity [10], impose additional overhead on transactional memory access operations to reduce validation overhead [5], or are imprecise and may lead to an unspecified number of false conflicts [12].

- We present and evaluate an algorithm that further reduces validation overhead by using bitmaps to represent information about transactional operations in a compressed form. Using bitmaps reduces both the space and time overheads of validation. This enhancement is important since even a linear validation scheme can incur a substantial overhead for a long running transaction. The STM presented by Welc *et al* [12] also uses bitmaps for validation, but they use this as the sole validation scheme. Therefore, to reduce the number of spurious aborts, their algorithm uses very large maps which negates any space savings, and can still lead to unpredictable performance degradation.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the existing validation algorithms in STMs with optimistic read concurrency and motivates the need for more efficient solutions. Section 3 describes a novel optimized validation algorithm linear in the number of reads executed by a transaction that is both precise and imposes no additional overhead on transactional data access operations. Section 4 presents a validation algorithm that compresses information about shared memory operations executed by a transaction into bitmaps. In section 5 we evaluate performance of the presented algorithms. Finally, section 6 contains the related work.

2. Overview

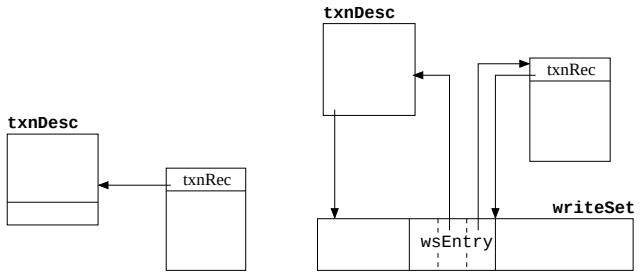
Our work is set in the context of an STM that implements optimistic read concurrency using version numbers, and pessimistic write concurrency using exclusive write locks [10, 5]. This type of STM has been independently proven to achieve high run-time performance [10, 5]. In this type of STM every access to a shared data item is mediated using a *transaction record* (called *STM Word* by Harris *et al* [5]). A transaction record may contain either a version number or a *lock pointer*. A lock pointer points to a data structure representing an exclusive lock and containing information about the lock owner.

If a transaction record associated with a given data item contains a version number, then other transactions are allowed to read this data item. Otherwise, only the lock owner is allowed to access this data item. Lock acquisition and release is controlled by the 2PL [2] protocol – locks are acquired when the transaction is active and released only upon transaction termination. Version numbers get incremented by the writers to indicate modifications of data items.

Every reader records a version number associated with a data item it is about to read in its local *read set*. A read set entry consists of a pointer to the transaction record associated with the data item being read and the recorded version number. A committing transaction validates its read operations by comparing the version numbers in its local read set with the current version numbers stored in the corresponding transaction records in the main memory. If all the version numbers match, all reads are considered valid and the transaction can commit.

Every writer records a version number associated with a data item it is about to write in its local *write set*. An entry in the write set, like that of a read set entry, consists of a pointer to the transaction record associated with the data item being written and the recorded version number. Version numbers recorded in the write set are used by the terminating transactions during the lock release procedure to restore transaction records to their "unlocked" state.

We have described the STM data structures at an abstract level to ease the presentation. We will dive into the details in the later sections when we describe the algorithms.



(a) McRT-STM (b) Bartok-STM

Figure 1. Data item locked by a transaction

2.1 Motivation

Processing of the read set during validation is a potential source of significant overhead. Ideally this overhead should be reduced as much as possible while, at the same time, keeping the cost of transactional data access operations low. Unfortunately, these two requirements are somewhat contradictory, as we will demonstrate through the analysis of solutions used by two highly efficient and well documented STMs, one by Harris *et al* implemented in the Bartok optimizing compiler [5] (from now on we will call it Bartok-STM to emphasize a distinction from Harris's previous STM implementation [3]) and the other, called McRT-STM, by Saha *et al* [10]. Both STMs use version numbers to validate correctness of read operations and exclusive locks for ensuring correctness of write operations, as described previously.

In both STMs a transaction can read or write a data item that has already been locked only if it already owns the lock. This *ownership test* must be implemented efficiently since it is executed for every transactional memory operation. The approach used in McRT-STM is illustrated in Figure 1(a). The transaction record (`txnRec`) associated with the locked data item points directly to the lock owner's *transaction descriptor* (`txnDesc`), which allows for direct identification of the lock owner. In Bartok-STM, for reasons described later in this section, the lock pointer points to an entry in the lock owner's write set. In order to facilitate the ownership test, in Bartok-STM an entry in the write set contains (in addition to a transaction record pointer and a version number) a pointer to the lock owner's transaction descriptor. As a result, in case of Bartok-STM, identification of the lock owner involves two levels of indirection instead of just one. The essence of the scheme used in Bartok-STM is illustrated in Figure 1(b). Transaction record points to an entry (`wsEntry`) in the lock owner's write set (`writeSet`)[1] whose middle slot contains a pointer to the lock owner's transaction descriptor. The remaining slots of the write set entry contain a back-pointer to the transaction record (right-most slot) and a version number (left-most slot) as described in Section 2. Additionally, in both STM-s both the read set and the write set of a given transaction are accessible via the transaction's descriptor (this information has been omitted from the figures for clarity).

The choice of the solution for the ownership test affects not only the efficiency of the data access operations but also the efficiency of the validation procedure. Consider a scenario where a transaction reads a data item (recording the version number), then locks the same data item and writes to it. The validation procedure must then verify that no other transaction modified this item between the read and the subsequent write operation. This is achieved by comparing the version number recorded (upon read) in the read set

[1] Only one entry in the write set is distinguished to improve clarity of the figure.

with a version number recorded by the same transaction (upon the following write) in the write set.

In Bartok-STM the time to perform this comparison is constant. The entry in the read set representing a locked data item points to the transaction record associated with this item while, as illustrated in Figure 1(b), the transaction record points to the lock owner's write set. Hence the version number recorded in the write set can be quickly recovered. The validation algorithm is then linear in the size of the read set. In McRT-STM the entry in the read set representing a locked data item also points to the transaction record associated with this item but the transaction record, as illustrated in Figure 1(a), points to the transaction descriptor representing the owner of the lock. The verification procedure needs in this case to scan the lock owner's write set (accessible through lock owner's transaction descriptor) to locate the version number for the appropriate data item. This results in an algorithm whose complexity is $O(M \times N)$ where M is the size of the read set and N is the size of the write set.

In summary, Bartok-STM features a fast validation procedure at a cost of an additional level of indirection during the ownership tests and an additional value that needs to be stored in every write set entry, whereas McRT-STM implements a much slower validation procedure but avoids a second level of indirection when testing lock ownership as well as additional write set space overhead. The first major contribution of our current work is a novel algorithm that combines the benefits of both algorithms described earlier in this section – its complexity is linear in the size of the read set, a single write set entry consists of only two slots and the additional level of indirection during ownership tests is avoided as well. However, even such an algorithm can incur a significant overhead for long running transactions, both in terms of validation time as well as the space overhead of maintaining the read set. Therefore we also present an algorithm that limits (and can potentially eliminate) the need for maintaining and processing a transaction's read set.

3. Optimized Linear Validation

We implemented the optimized validation algorithm in McRT-STM and compared it with the original validation algorithm described in Section 2. In order to maintain the low cost of the lock ownership test, the new algorithm guarantees that it is sufficient for a transaction accessing a data item to only inspect the contents of the transaction record to determine the lock ownership (no additional levels of indirection are involved). At the same time, it avoids multiple searches of the write set during validation. [2] The new algorithm works by temporarily modifying the contents of transaction records during the validation procedure to allow easy access to the version numbers recorded (upon write) in the write set.

3.1 Algorithm overview

In the original validation algorithm, a transaction record could only contain either a version number or a lock pointer (pointing to a transaction descriptor) at any given time. These two cases were distinguished by checking a designated low-order *version bit* of the value stored in a transaction record. A transaction record contained a version number if the bit was set, and contained a lock pointer (indicating that the data item is locked) if the bit was cleared. We took advantage of this property when developing the new optimized validation algorithm.

In the new algorithm, when a data item is unlocked the transaction record associated with this data item contains the version number and the version bit is set. When a data item is locked, however, its transaction record can either contain a pointer to the lock owner's transaction descriptor (when the corresponding transaction is active) or a pointer to an entry in the lock owner's write set (when lock owner's transaction is being validated). In both these cases the version bit is unset. Since the meaning of the version bit remains the same when a transaction is active, the access operations on the shared data do not have to change and, as a result, remain as efficient as in the original algorithm. Any data access operation of a given transaction can proceed if the bit is unset since the item is unlocked. If the bit is set, the operation of a given transaction can proceed only if the contents of the transaction record points to its own transaction descriptor. Otherwise, the transaction record points to either another transaction's descriptor, or to an entry in another transaction's write set. We would like to emphasize that, unlike in the approach adopted by Bartok-STM, a transaction record is only made to point to an entry in the write set for the duration of the validation procedure, and then only if the transaction reads a variable after writing into it. This is important since an optimizing compiler can statically detect and eliminate many or even all write-after-read situations. In such a case, our algorithm neither incurs an overhead during transactional access operations, nor during the validation procedure.

[2] These searches were necessary in the original validation algorithm present in McRT-STM to determine the validity of accesses to data items that have been first read and then written by the same transaction.

```
bool VALIDATE(TxnDescriptor *txnDesc) {
 // ************ PHASE 1 ************
 FOREACH rsEntry IN txnDesc->readSet {
  IF (*(rsEntry->txnRec) != rsEntry->version) {
   // version numbers do not match
   IF ((*(rsEntry->txnRec) & ~VALIDATE_MASK) == txnDesc) {
    // mark write after read
    *(rsEntry->txnRec) = txnDesc | VALIDATE_MASK;
   }
   ELSE RETURN FALSE; // validation failure
 }}
 // if no write after read has been performed
 // then PHASE 2 and PHASE 3 can be skipped

 // ************ PHASE 2 ************
 FOREACH wsEntry IN txnDesc->writeSet {
  IF (*(wsEntry->txnRec) & VALIDATE_MASK)
   *(wsEntry->txnRec) = wsEntry | VALIDATE_MASK;
 }
 // ************ PHASE 3 ************
 FOREACH rsEntry IN txnDesc->readSet {
  IF ((*(rsEntry->txnRec) & ~VALIDATE_MASK) == txnDesc &&
      POINTS_TO_WRITE_SET(*(rsEntry->txnRec)), txnDesc) {
   // transaction record  points to txnDesc's write set;
   // compare version numbers in both sets
   IF ((*(rsEntry->txnRec))->version != rsEntry->version)
    RETURN FALSE;
 }}
 RETURN TRUE;
}
```

Figure 2. Optimized linear validation procedure

3.2 Validation Procedure

During transaction's validation procedure we need to be able to distinguish transaction records associated with data items that have been both read and written, and are currently locked by the validating transaction. While the validation procedure is in progress, these transaction records can only point either to the validating transaction's descriptor or to an entry in the validating transaction's write set. Since pointers are aligned we designate another low-order bit (called *validation bit*) for this purpose.

The validation procedure consists of the following phases. Phases 2 and 3 are optional and depend on the result of executing phase 1.

1. **Validate and Mark**: For every entry in the read set of the validating transaction inspect the contents of the transaction record it points to. If the transaction record contains the same version number as the one recorded in the read set, proceed to the next read set entry (the read is valid). If the transaction record contains the validating transaction's descriptor (with the validation bit set or unset) then the validating transaction both read and wrote the same data item. Set the validation bit (if not yet set) and proceed to the next read set entry. In all other cases the validation procedure fails. We do not execute the subsequent phases if the transaction did not have a write-after-read situation, or the validation failed.
2. **Redirect**: For every entry in the write set of the validating transaction inspect the contents of the transaction record it points to. If the validation bit of the value stored in the transaction record is set, redirect the transaction record to point to the currently inspected entry in the write set (keeping the validation bit set).
3. **Verify**: For every entry in the read set of the validating transaction inspect the contents of the transaction record it points to. If the validation bit of the value stored in the transaction record is set and the transaction record contains a pointer to an entry in the validating transaction's write set, compare the version number stored in the read set with the version number stored in the write set. If the values are equal, proceed to the next read set entry; otherwise the validation procedure fails. Please note that the write set entry in McRT-STM, unlike the write set entry in Bartok-STM, consists of only two slots. Therefore, in case a transaction record points to a write set entry of some lock owner (not necessarily the one being validated), we need an alternative solution for lock owner identification. [3] We take advantage of the fact that in McRT-STM write set is composed of memory chunks aligned on a fixed-size boundary. This allows us to store a pointer to the lock owner's transaction descriptor at a fixed offset from the base of each chunk and access it using simple pointer-arithmetic operations.

The pseudo-code describing the new linear validation algorithm is presented in Figure 6 with all three phases explicitly distinguished. The `VALIDATE_MASK` constant represents a mask used to mask out the validation bit. The `POINTS_TO_WRITE_SET` procedure summarizes actions (bit operations and the lock owner identification test) required to determine if the value stored in the transaction record points to an entry in the write set of the transaction executing the validation procedure.

Please note that if the validation procedure is executed upon transaction termination, the clean-up of transaction records can be piggy-backed on the lock release procedure which has to be executed regardless of the validation procedure's result. Only if validation is performed while the transaction is in progress (*eg* , for periodic validation) the explicit cleanup (involving an additional pass over the write set) is required. It is also important to observe that even though the complexity of the algorithm is $O((M \times 2) + N)$ in the worst case (M is the size of the read set and N is the size of the write set), the additional passes through the read set and the write set are only necessary if the validating transaction read and then subsequently wrote to some data items it has accessed. If it is possible to detect these kinds of situations it is beneficial to eagerly acquire a write lock at the point of read to guarantee that the data item is not going to be modified by a different transaction between the read and the write. If all such cases are detected, then the complexity of the new validation algorithm is reduced to $O(M)$.

[3] In Bartok-STM a pointer to lock owner's descriptor is stored directly in the third slot of the write set.

A compiler analysis performing this task has been described by Adl-Tabatabai *et al* [1] and Harris *et al* [5].

4. Bitmap-based Validation

Similarly to the optimized linear validation procedure, the bitmap-based algorithm has been implemented in the context of McRT-STM. The original validation procedure used in McRT-STM requires processing of the entire read set (of the size equal to the number of reads performed by the validating transaction) to detect conflicts with respect to updates performed by other transactions. A general idea behind the bitmap-based validation algorithm is to coarsen the granularity of conflict detection by compressing information recorded in the read set into a form of a bitmap, optimizing the validation procedure towards mostly-read-only transactions. Bitmaps are used to detect conflicts only on the first attempt to commit a transaction and, upon validation failure and transaction abort, the original validation procedure is used on the following commit attempts of the re-executed transaction. Unlike the optimized linear validation algorithm described in Section 3, the bitmap-based algorithm requires modification of the code fragments implementing transactional read and write operations, called *read and write barriers*.

The bitmap-based algorithm assumes that every eligible transaction (as described below, bitmaps should not be used to detect conflicts under certain circumstances) initially uses bitmaps to detect conflicts and only upon failed validation and a resulting abort uses the original validation procedure during the subsequent re-execution. The write barriers execute both the code required to support the original validation procedure and the code implementing the algorithm using bitmaps. This is required because enough information must be maintained at all times to facilitate conflict detection with respect to other transactions that may be using different versions of the validation procedure at the same time. The read barriers use a conditional to decide which version of the validation procedure is currently used and to choose the appropriate sequence of code to execute. The expected benefit is the reduction of overheads related to both the time required to execute the read barriers and the validation procedure itself (no need to either maintain or process a read set) at a cost of potentially increasing both the number of aborts and the cost of the write barrier. A detailed description of the bitmap-based algorithm is presented below.

4.1 Overview

We use two transaction-local bitmaps (*local read map* and *local write map*) to record reads and writes performed by the transaction to all shared data items. We currently use maps of the 64-bit size to achieve high efficiency of operations used to manipulate bitmaps. Every slot (bit) in the bitmap represents access to a single shared data item. A slot in the map corresponding to the data item being accessed is computed using the address of the data item. In case the address of a data item accessed inside of a transaction using the bitmap-based algorithm changes while this transaction is still active (*eg* , as a result of a copying garbage collector moving objects around), all currently active transactions using the same scheme must be aborted.

A *global write map* is maintained to record updates performed by all transactions executing concurrently. Information about all updates performed by a successfully committed transaction (stored in its local write map) is merged with the information currently represented in the global write map (initially empty). Additionally, a transactions counter is maintained to represent the number of concurrently executing transactions. Merging of local and global maps needs to happen only if the counter is greater than one. Transactional reads are validated by computing an intersection of the local read map that belongs to the committing transaction and

```
void BEGIN(TxnDescriptor *txnDesc, int64 globalMap) {
 IF (txnDesc->useBitmaps) {
  WHILE(TRUE) {
   int tCount = globalMap & COUNT_MASK;

   IF (tCount == 0)                  // RESET MAP
    IF (UPDATE_MAP_CAS(0, tCount+1)) BREAK;
   ELSE IF (tCount < COUNT_MAX &&   // INCREMENT COUNT
            (globalMap & MAP_MASK) == 0)
    IF (UPDATE_MAP_CAS(globalMap, tCount+1)) BREAK;
   ELSE {                            // DISABLE BITMAPS
    txnDesc->useBitmaps = false;
    ORIGINAL_BEGIN(txnDesc);
    BREAK;
 }}}
 ELSE ORIGINAL_BEGIN(txnDesc);
}
```

Figure 3. Transaction begin

the global write map. If the intersection is empty, validation is successful, otherwise it fails and the transaction is aborted. As mentioned previously, in some situations all transactions using bitmap-based scheme may have to be aborted. This can be easily handled by setting all the bits in the global map to one.

Bitmaps should be used to detect conflicts only if the number of concurrently executing transactions is relatively low (otherwise there is a high chance of quickly filling up the global write map which would lead to conflicts being detected often). This allows us to set the maximum value of the transactions counter to a low value (currently 32), and to implement it using the (5) lowest bits of the global write map. Updates to the counter and the global write map can then be performed atomically using a single 64-bit CAS (atomic compare-and-swap) operation.

4.2 Transaction Begin

If at the start of a transaction the transactions counter is equal to zero then the transaction clears the global write map, increments the counter and proceeds. If the transactions counter is greater than zero (another transaction is already executing) and either the global write map is non-empty or the maximum value of the counter has been reached, then the starting transaction uses the original validation procedure (a separate flag is used to maintain this information – in such case only the original validation procedure is attempted at commit time). In all other cases, no action is required and the transaction is free to proceed.

Falling back to the precise validation procedure in case of non-empty global write map is required to avoid a problem of never clearing the map in case the number of concurrently executing transactions is high (in particular, if a new transaction is often

```
void COMMIT(TxnDescriptor *txnDesc, int64 globalMap) {
 IF (txnDesc->useBitmaps) {
  int tCount = globalMap & COUNT_MASK;
  int64 mapIntersect = txnDesc->readMap & globalMap;

  // UPDATE MAP BOTH ON COMMIT AND ON ABORT
  UPDATE_MAP_CAS(globalMap | txnDesc->writeMap, tCount  1);
  IF (mapIntersect != 0) {
   // USE ORIGINAL ALGORITHM ON RE-EXECUTION
   txnDesc->useBitmaps = false;
   ABORT();
 }}
 ELSE ORIGINAL_COMMIT(txnDesc);
}
```

Figure 4. Transaction commit

```
int64 TAG_MAP(int64 localMap, int *addr) {
 int slot = (addr >> BITS_IGNORED) % MAP_SIZE;
 int64 slotMask = FIRST_MAP_SLOT << slot;
 RETURN localMap | slotMask;
}

int READ_BARRIER(TxnDescriptor *txnDesc,int *addr) {
 IF (txnDesc->useBitmaps)
  txnDesc->readMap = TAG_MAP(readMap, addr);
 ELSE ORIGINAL_READ_BARRIER(txnDesc, addr);
 RETURN *addr;
}

void WRITE_BARRIER(TxnDescriptor *txnDesc,
                   int *addr, int newValue) {
 txnDesc->writeMap = TAG_MAP(writeMap, addr);
 ORIGINAL_WRITE_BARRIER(txnDesc, addr);
 *addr = newValue;
}
```

Figure 5. Read and write barriers

started before all the previously executed ones are committed). In other words, we use bitmaps to detect conflicts only for transactions that are started before the first update of the global write map.

The pseudo-code representing the transaction begin operation is presented in Figure 3. The `useBitmaps` flag is used to determine if bitmaps should be used at all for validation purposes. Procedure `UPDATE_MAP_CAS()` is an abbreviation for an operation using a single CAS to update both the contents of the map (passed as the first parameter) and the counter (passed as the second parameter) – it returns `TRUE` upon successful update and `FALSE` upon failure. The `COUNT_MASK` and `MAP_MASK` constants are used to mask the appropriate bits of the word representing the global map to extract the transactions counter and the contents of the map, respectively. The `COUNT_MAX` constant represents the maximum counter value (fitting into the specified number of bits).

4.3 Transaction Commit

In case bitmaps are used to detect conflicts (as indicated by the appropriate flag), the committing transaction computes the intersection of its local read map and the global write map to validate correctness of its read operations.

If the intersection is empty then the validation succeeds, otherwise the committing transaction is aborted and re-executed. In both cases, the committing transaction merges contents of its local write map with the contents of the global write map and decrements the transactions counter. The merging of write maps is required in both cases since the updates performed by the committing transaction have already been reflected in the main memory and other transactions need to be notified about the possibility of a conflict. The pseudo-code representing transaction commit operation is presented in Figure 4. Note that upon failure of the procedure using bitmaps the flag to disable optimized procedure is set and the transaction is aborted to be re-executed using the original algorithm.

4.4 Barriers

The read and write barriers, presented in Figure 5, have been modified to implement tagging of the local maps. The `TAG_MAP()` procedure is shared between all the barriers. We found out experimentally that in order to achieve the best distribution of values in the map we should ignore a certain number of bits in the address (represented by the `BITS_IGNORED` constant) when computing the map slot number. The `FIRST_MAP_SLOT` constant represents the first available slot number (determined by how many low order bits are used to store the transactions count). Note that in the case of the read barrier, it is either the code fragment supporting the

% reads	hash-table		b-tree	
	1 CPU	16 CPUs	1 CPU	16 CPUs
10	0.9991	0.9907	1.0229	0.9545
30	0.9920	0.9745	1.0238	0.9048
50	1.0006	0.9577	1.0178	1.2029
70	0.9975	0.9758	1.0102	0.9501
90	0.9943	0.9667	0.9856	0.9624

Figure 6. Execution times

original procedure or the code fragment supporting algorithm using bitmaps that gets executed. In case of the write barrier the code supporting both styles of validation must be executed.

5. Performance Evaluation

In this section we will present results of the performance evaluation of both the new optimized linear validation algorithm and the bitmap-based validation algorithm. Both solutions are implemented in the context of McRT-STM and evaluated on the Intel Xeon™ 2.2 GHz machine with 16 CPUs and 16 GB of RAM running Red Hat Enterprise Linux Release 3 (with 2.4.21-28.ELsmp kernel). Both new algorithms are being compared with the original validation algorithm present in McRT-STM. The evaluation is based on measuring execution times and other execution metrics (such as aborts) of two transaction-enabled benchmarks implementing accesses to common data structures: hash-table and b-tree.

5.1 Optimized Linear Validation

The first set of numbers verifies that in the common case the performance of the new linear validation algorithm does not negatively affect the overall performance of the system. We first executed both benchmarks using one thread running on a single CPU to model workloads that do not exhibit any contention. We also executed both benchmarks using 16 threads running on 16 CPUs, even though we did not expect the contention to have any effect on the performance of the algorithm. In both uncontended and contended cases we varied the percentage of reads (vs. the percentage of writes) from 10% of reads to 90% of reads (the total number of operations performed by all threads was 2^{17}). We summarize the results obtained for both the hash-table benchmark and the b-tree benchmarks in Table 6. The table reports execution times for the new linear algorithm normalized with respect to the execution times of the original algorithm.

The analysis of the numbers presented in Table 6 confirms that in the common case the performance of both algorithms is comparable. In the case of single-threaded execution the difference in performance rarely exceeds 1%. We believe that the larger differences observed during multi-threaded runs is due to the non-determinism

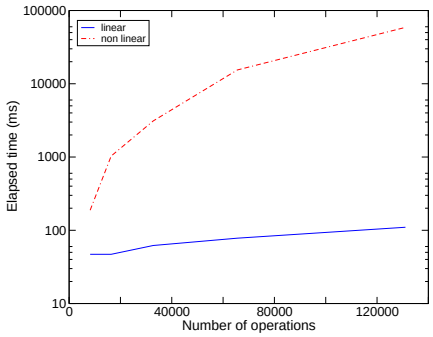


Figure 7. Execution times – linear algorithm vs. non-linear algorithm

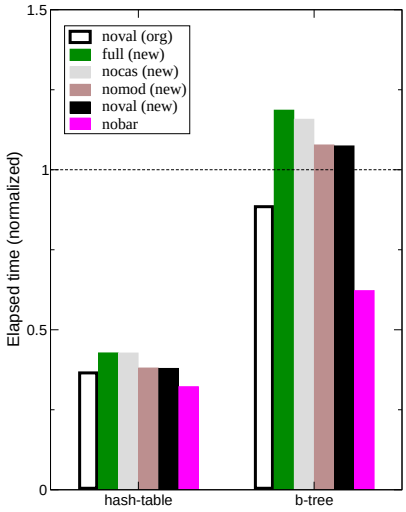


Figure 8. Overhead split – 1 thread, 1 CPU

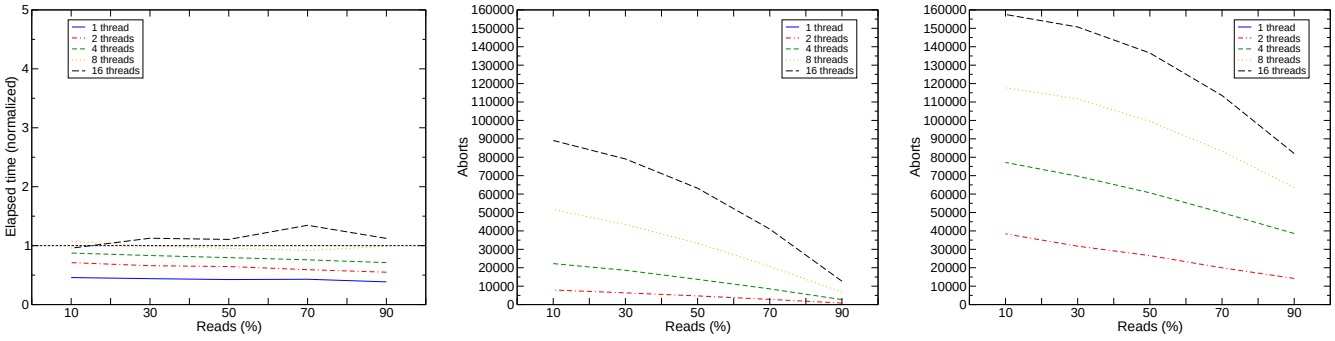
in the results we obtained (which we were unable to fully eliminate despite averaging over 25 runs).

Despite both algorithms performing similarly in the common case, the performance of the original validation algorithm is expected to degrade quite dramatically, dropping exponentially as the number of write-after-read occurrences grows. At the same time, the execution time of the new validation algorithm is expected to only grow linearly in the number of operations. In order to confirm this hypothesis, we measured execution times of a microbenchmark that pulls every entry from a given pre-initialized data structure (hash-table) and then both reads and writes a value stored in that entry. We vary the number of entries (and thus operations performed) from 2^{13} to 2^{17} and plot the execution times (total execution time of the micro-benchmark in microseconds using a logarithmic scale) in Figure 7. As we can observe, our hypothesis indeed holds – performance of the new algorithm remains linear while the performance of the original algorithm rapidly degrades as the number of write-after-read operations increases.

5.2 Bitmap-based Validation

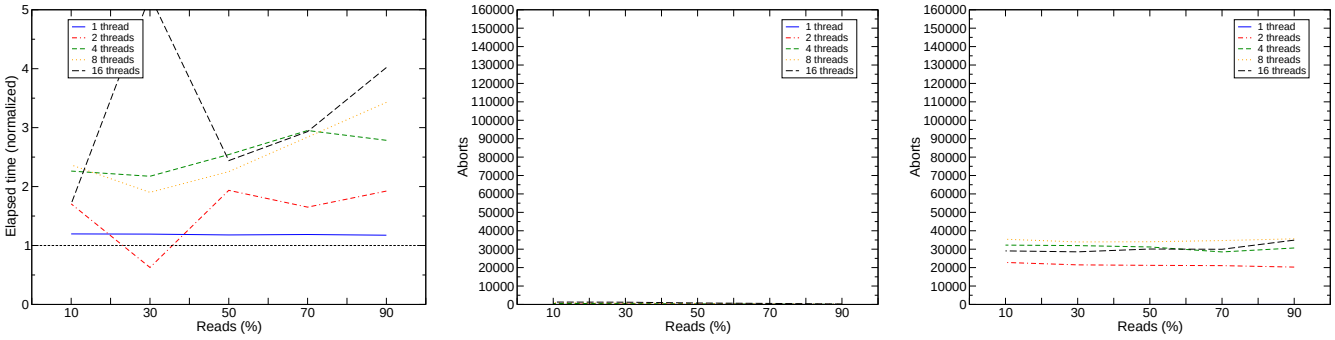
The first set of numbers represents an "ideal" case for the bitmap-based validation procedure – both benchmarks were executed within a single thread on a single physical CPU. As a result no aborts could ever occur. Additionally we measured impact of different implementation components on the overall performance. Figure 8 describes execution times for both benchmarks, differentiating between separate components (represented by different bars). All execution times are normalized with respect to the "baseline" case, that is execution times for a version of the system that uses the original validation procedure only.

The first bar is an exception – it represents execution time for the configuration using the original validation procedure where the *verification step* (part of the validation procedure performed either at commit time or during periodic validation) has been omitted. In other words, it constitutes our ideal target in terms of expected performance – our major goal is to reduce overheads related to performing the verification step. This bar represents one of two major components contributing to the overheads related to supporting the original (and in fact any other) validation procedure – the other one is related to work performed during data accesses (in the barriers). The last bar represents execution time where in addition to omitting the verification step, both read and write barriers have



(a) execution times (b) original aborts (c) bitmap-based aborts

Figure 9. Hash-table benchmark



(a) execution times (b) original aborts (c) bitmap-based aborts

Figure 10. B-tree benchmark

also been removed (and all the data access operations turned into "regular" loads and stores). The difference between these two bars represents the cost of the barriers. Please note, that the execution time described by the last bar represents both algorithms (original and bitmap-based) – by omitting the verification step as well as the barriers these versions of the system have been effectively unified.

The second bar represents a fully functional system supporting the bitmap-based validation procedure (the remaining bars, even though correctly executing their respective workloads in the single-threaded case could behave incorrectly in presence of multiple concurrent threads). The third bar represents a version of the algorithm where the CAS operations used to update the global map and the transactions counter have been replaced by regular writes. The fourth bar represents a version of the algorithm where the modulo division operation used to compute the appropriate slot in the map has been replaced (in addition to removing CAS operations) with a bit-wise AND operation extracting the n lowest bits of the address. Using the n lowest bits for slot computation is unlikely to provide a good distribution of slots in the map – in case of real-life workloads it might be difficult to accept. The fifth bar represents a version of the algorithm where the configuration represented by the previous (fourth) bar has been further modified to completely omit the verification step of the bitmap-based procedure (similarly to the verification step for the original procedure being removed in case of the first bar). The sixth (and last) bar has already been described.

The first conclusion that can be drawn from the analysis of Figure 8 is that we can expect high performance gains from optimizing the validation step only in the case of the hash-table benchmark, as indicated by the size of the first bar. It is a direct result of both a large number of locations read by every transaction executed in this benchmark (orders of magnitude more than for the b-tree benchmark) and periodic validations performed by transactions for all locations they have read.

The first bar, as mentioned before, represents optimization potential. The difference between the height of the first bar and the height of the fifth bar represents the difference in the barrier cost between the original scheme and the optimized scheme. As we can see the cost of the barriers in the bitmap-based validation procedure is larger than for the original even in the case when modulo division has been replaced by a bit-wise AND. As a result, when the optimization potential is small (as is the case of the b-tree benchmark) no performance improvement is observed.

The second set of numbers attempts to answer the question of how both the bitmap-based validation procedure behaves under contention. Figure 9(a) and Figure 10(a) plot execution times for the hash-table benchmark and the b-tree benchmark, respectively. The graphs reflect execution times for the bitmap-based scheme normalized with respect to execution times using the original validation procedure under varying levels of contention: 1–16 threads running on 1–16 CPUs respectively, and percentage of reads (vs. the percentage of writes) varying from 10% of reads to 90% of reads. Let us first analyze performance of the hash-table benchmark (Figure 9(a)). The bitmap-based procedure outperforms the original algorithm for configurations running up to 8 threads on 8 CPUs, but starts performing worse than the original algorithm at the highest level of contention (when 16 threads on 16 CPUs are being run). As we can observe, the increase in the number of aborts for the configurations using the bitmap-based validation procedure (plotted in Figure 9(c) seems to have relatively small impact on the overall performance. The performance of the bitmap-based al-

gorithm is better than that of the original one for all but the most contended configuration despite the number of aborts being quite dramatically increased with respect to the numbers for the original validation procedure (plotted in Figure 9(b). We believe that the explanation for this behavior lies in periodic validations of all the data read by transactions executed in this benchmark. The cost of periodic validation is very low when using the bitmap-based version of the procedure (a transaction intersects the maps and either proceeds or aborts) whereas in the case of the original algorithm traversal of the entire read set is required. As a result, the cost of a (failed) transactional execution is potentially much higher in the case of the original algorithm.

The analysis of the execution times for the b-tree benchmark is much simpler. The bitmap-based procedure is unable to outperform the original algorithm (with the exception of execution anomalies – explanation is presented below) as indicated on Figure 10(a). Also, the increase in the number of aborts has the expected result on performance. Aborts for both types of algorithms are plotted on Figure 10(b) (original validation procedure) and Figure 10(c) (bitmap-based validation procedure) with the number of aborts for the original algorithm being extremely small.

When running the benchmarks, similarly to the experiments we performed for the new optimized linear algorithm, we observed a degree of variability between different runs of the same benchmark (observed over 25 runs of every configuration) and a significant amount of jitter. This effect is much more visible in the case of the b-tree benchmark whose running time is short – we believe that benchmark runs where the bitmap-based scheme performed better than the original one are a direct result of this behavior. Therefore, we repeated our measurement for the b-tree benchmark with the number of iterations increased by the factor of four to increase the execution times. Unfortunately the variability has not been eliminated but the overall conclusion remains the same – no performance benefit can be expected from running any of the b-tree benchmark configurations when using the bitmap-based validation procedure.

One of the conclusions that can be drawn from the analysis of these results is that adding any additional overheads to the read and write barriers should be avoided when implementing similar solutions. The increased barrier cost dwarfed any potential performance improvement of the verification step for the b-tree benchmark. A more general conclusion is that while the bitmap-based validation procedure has potential to improve the performance over the original algorithm, it is not suitable for a general purpose system in its current form as it performs well only under selected workloads (large number of reads, potentially in conjunction with frequent periodic validations) and may degrade performance when used as the primary validation procedure. At the same time, the approach presented could prove itself useful in case the validation procedure must be invoked much more frequently (*eg* , at every data access operation, in the extreme case). In such case, the additional overheads described above could appear to be more tolerable.

6. Related Work

A linear validation algorithm has been proposed by Harris *et al* [5] and implemented in the context of their STM system. However, in their solution transaction records associated with locked data items always (throughout the entire lifetime of the lock owner) point to the appropriate entry in the lock owner's write set. As a result, and unlike in our own linear algorithm, the ownership tests performed during transactional memory access operations require an additional level of indirection.

Bitmaps have been previously used to support validation algorithm by Welc *et al* [12]. In their system, however, the validation procedure using bitmaps is the primary one. As a result, in order to achieve a reasonable precision of the validation procedure, they must use large bitmaps (16,000 slots) which may have negative impact on both the memory footprint as well as on the time it takes to execute the validation procedure. Additionally, imprecision resulting from the (exclusive) use of bitmaps for validation may result in repeated transaction aborts, whereas in our system, a transaction can be aborted only once as a result of using information about reads in a compressed form (subsequent re-executions use the original validation algorithm).

References

[1] ADL-TABATABAI, A.-R., LEWIS, B. T., MENON, V., MURPHY, B. R., SAHA, B., AND SHPEISMAN, T. Compiler and runtime support for efficient software transactional memory. In *PLDI 2006*.

[2] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. Data Management Systems. Morgan Kaufmann, 1993.

[3] HARRIS, T., AND FRASER, K. Language support for lightweight transactions. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Anaheim, California, Nov.). *ACM SIGPLAN Notices 38*, 11 (Nov. 2003), pp. 388–402.

[4] HARRIS, T., MARLOW, S., PEYTON-JONES, S., AND HERLIHY, M. Composable memory transactions. In *PPoPP 2005*.

[5] HARRIS, T., PLESKO, M., SHINNAR, A., AND TARDITI, D. Optimizing memory transactions. In *PLDI 2006*.

[6] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, III, W. N. Software transactional memory for dynamic-sized data structures. In *PODC 2003*.

[7] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture* (San Diego, California, May). 1993, pp. 289–300.

[8] MARTÍNEZ, J. F., AND TORRELLAS, J. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *ASPLOS 2003*.

[9] RAJWAR, R., AND GOODMAN, J. R. Transactional lock-free execution of lock-based programs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, Oct.). *ACM SIGPLAN Notices 37*, 10 (Oct. 2002), pp. 5–17.

[10] SAHA, B., ADL-TABATABAI, A.-R., HUDSON, R. L., MINH, C. C., AND HERTZBERG, B. A high performance software transactional memory system for a multi-core runtime. In *PPoPP 2006*.

[11] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *PODC 1995*.

[12] WELC, A., HOSKING, A. L., AND JAGANNATHAN, S. Transparently reconciling transactions with locking for Java synchronization. In *ECOOP 2007*.