COMPUTING
SCIENCE

Introducing Semantic Conflict Resolution to Word Based Software Transactional Memory

Craig Sharp and Graham Morgan

TECHNICAL REPORT SERIES
No. CS-TR-1409 February 2014
Introducing Semantic Conflict Resolution to Word Based Software Transactional Memory

C. Sharp and G. Morgan

Abstract

In this paper we describe the implementation of a contention manager which deals with semantic conflicts within word based Software Transactional Memory. Semantic conflicts are introduced which prevent transactions from committing their execution due to the presence of some application condition and can be resolved by the reordering of transaction execution. We consider application semantics to be just as important as linearizability in deriving a suitable execution strategy for the progress of the application. The benefit of our approach is demonstrated with benchmarked performance figures demonstrating the ability of our technique to address semantic conflicts within the context of two realistic scenarios in comparison to existing contention management policies.
Abstract

In this paper we describe the implementation of a contention manager which deals with semantic conflicts within word based Software Transactional Memory. Semantic conflicts are introduced which prevent transactions from committing their execution due to the presence of some application condition and can be resolved by the reordering of transaction execution. We consider application semantics to be just as important as linearizability in deriving a suitable execution strategy for the progress of the application. The benefit of our approach is demonstrated with benchmarked performance figures demonstrating the ability of our technique to address semantic conflicts within the context of two realistic scenarios in comparison to existing contention management policies.

About the authors

Craig Sharp is a research associate in the School of Computing Science at Newcastle. Craig's research interests include concurrency control and highly parallel computing.

Graham Morgan is a senior lecturer in the School of Computing Science at the Newcastle University. Graham leads the Game Engineering MSc at Newcastle and carries out research in the areas of systems and simulation. His interests are in the area of distributed applications, including web services, networked virtual environments, fault tolerance and group communications.

Suggested keywords

TRANSACTIONAL MEMORY
CONTENTION MANAGEMENT
SEMANTIC CONFLICTS
Abstract

In this paper we describe the implementation of a contention manager which deals with semantic conflicts within word based Software Transactional Memory. Semantic conflicts are introduced which prevent transactions from committing their execution due to the presence of some application condition and can be resolved by the reordering of transaction execution. We consider application semantics to be just as important as linearizability in deriving a suitable execution strategy for the progress of the application.

The benefit of our approach is demonstrated with benchmarked performance figures demonstrating the ability of our technique to address semantic conflicts within the context of two realistic scenarios in comparison to existing contention management policies.

Categories and Subject Descriptors D.1.3 [Programming Techniques]: Concurrent Programming

General Terms Algorithms, Performance

Keywords Transactional Memory, Contention Management, Semantic Conflicts

1. Introduction

Software Transactional Memory (STM) has become a popular research area for concurrent programmers given that the STM abstraction offers ease of use in comparison to lock based approaches. More powerfully, composing sections of concurrent code can be achieved with ease using STM unlike a lock-based implementation [13], while the generic interface of STM is easily understood.

At the time of writing, there exist a variety of STM implementations with two approaches gaining prominence: object based and word based. Object based STMs [16, 18, 22] are generally particular to object orientated languages and represent shared data in the form of atomic objects. Conversely, shared data in word based STMs [7, 9, 10] is represented at the level of memory words.

Felber et al observed in [10] that word-based STMs allow transactional accesses to be mapped directly to the underlying memory system. As a result, word-based STMs offer: (i) easier integration into existing programming languages and (ii) greater efficiency in the context of compiler support. To this goal, TinySTM [10, 11] has been provided as a lightweight and efficient word-based STM. The (relatively) small and manageable code base makes TinySTM particularly attractive for word-based STM development, and can be easily integrated with the STAMP [19] benchmark suite. For these reasons, the developments in this paper have been integrated into TinySTM.

A significant feature of any STM system concerns the handling of aborted transactions, particularly when contention for shared resources is high. This task is typically delegated to the Contention Management Policy (CMP) whose role it is to decide the losing transaction whenever two transactions conflict. Various CMPs use different properties of a transaction to decide on a winner. Time-stamp for instance, is a CMP which gives priority to the transaction that began first. From the perspective of the CMP, there are concurrent conflicts, but from the perspective of the programmer, conflicts can be classified as two types: concurrency conflicts and semantic conflicts. Whereas concurrency conflicts are generated by the inter-thread interference of reads and writes on shared data, semantic conflicts can be conceived as conditions within the application which prevent a transaction from committing.

Ignoring the presence of semantic conflicts (when a thread explicitly aborts its own transaction) is potentially very damaging for the progress of the application. The simple scenario shown in Figure 1(A) exemplifies the problem where two threads execute a withdrawal and deposit transaction respectively. Let us suppose that there is a concurrent conflict between the withdrawer and depositor transactions and that the CMP decides to abort the depositor. If the withdrawer is waiting for the deposit to be made then it must explicitly abort, and both transactions will have to re-execute until the withdrawer is lucky enough to precede the depositor (or the CMP aborts the depositor).

Furthermore, this problem can be exacerbated if a CMP is used which judges transactions by their starting time or the amount of work completed. It is possible that the withdrawer may always succeed in aborting the depositor if the withdrawer began before the depositor or has carried out more work.

One superficial solution to the problem of semantic conflicts is to allow a ‘semantic transaction’ to commit rather than aborting explicitly (assuming no concurrent conflicts have occurred). The application may then reschedule the transaction to execute at some future time. In dealing with a semantic conflict by allowing the transaction to commit and re-execute, observe that the semantic issue has not been solved. Instead, the user must now decide when to re-execute semantic transactions and further unnecessary transactions will be created leading to greater contention within the application.

A semantic conflict may also be addressed with primitives provided by the STM system if available. Harris et al [13] for instance, provided primitives (retry, orElse) which allow a limited degree of transactional coordination, thus allowing the STM system to deal with the semantic conflict by imposing an ordering of transaction execution. The use of primitives to deal with semantic conflicts should also be discouraged, given that this places the burden on the application programmer for devising a correct coordination of transaction execution. If one considers the principle benefits of transactional memory over locking, absolving the application programmer of the duty of coordination is one such aim.
This paper describes an implementation of a CMP (Hugh2) which is designed to resolve semantic conflicts by exploring multiple permutations of parallel transaction execution. We consider a semantic conflict as the explicit abortion of a transaction, by its own thread, due to some condition in the shared data of the application. All STMs allow transactions to explicitly abort, therefore producing semantic conflicts requires no special features to be added to an STM. As any program which uses an STM can produce semantic conflicts, we suggest that our work is significant considering the implications for performance if semantic conflicts are ignored.

The remainder of this paper is organised as follows: Section 2 describes the Implementation. Section 3 provides a summary of Related Work to provide the context of our contribution and Section 4 provides an Evaluation of performance results obtained from the implementation of Hugh2. Finally, Section 5 concludes the paper and summarises possible avenues for future work.

2. Implementation

Hugh2 has been implemented as part of TinySTM (version 1.0.4). Our work provides contention management specifically for semantic conflicts, so any existing CMP can be used for concurrent conflicts. Numerous changes were necessary however to incorporate Hugh2, specifically, we enabled replication of transactions and introduced a new session-locking procedure.

Using session locks enables threads to share locks during the same session, but this feature requires us to use the write-back mode of locking as opposed to write-through which must enforce exclusivity of any locks held given that modifications are made directly to memory. Encounter-time (or eager) locking of shared data is used, although this is not a requirement of the technique.

2.1 Overview

Figure 1. Scenarios A and B contrast the approaches of a conventional CMP with the Hugh2 CMP when a semantic conflict occurs.

Hugh2 CMP is activated once some thread encounters a semantic conflict which has caused thread to explicitly abort its transaction. Before the aborted transaction is restarted, thread sets an internal flag to indicate that it has entered a session mode.

Once in session mode, thread re-executes its own transaction in addition to the transactions of any other threads currently in session mode. Each session mode thread executes a permutation of transactions, in an attempt to discover a transaction schedule which can resolve the semantic conflict(s). Figure 1(B) for example, shows the resolution of a semantic conflict that has occurred in a banking scenario involving two transactions accessing a single account (specifically, a depositor and withdrawer transaction). Thread 2 executes a permutation which succeeds in resolving the semantic conflict (the deposit ensures that the withdrawal can occur).

When there are no further transactions to execute, each thread performs consensus to determine whose permutation of transaction execution will be committed. Consensus is managed using a Universal Construction (hereafter UC). The UC is essentially a linked-list, which may be concurrently appended to by threads engaged in session mode. Each new node appended to the UC identifies the transactions that have been committed during a particular session.

Once a session has terminated, each participating thread can determine whether its own transaction was committed or aborted by reading the log of the Universal Construction. Those threads whose transactions remain uncommitted perform a new session, while the threads of the committed transactions return to non-session mode.

2.2 Aims and Contribution

In this paper we describe a CMP which incorporates some novel design features in order to resolve semantic conflicts in word based STM. In a previous paper we introduced a version of our CMP which worked with an object-based STM [] and discussed the implications of semantic conflicts. We provided several micro-benchmarks to assess the performance of our technique in small-scale scenarios. In this paper we describe significant changes required to bring our system into a state-of-the-art word-based STM and provide more robust testing using large-scale benchmarks (including the vacation scenario from the STAMP benchmark suite [19]). The aims and contributions are as follows:

- To adapt TinySTM with transaction replication and session locking to provide a CMP where semantic conflicts are resolved in a process both decoupled from the programmer and compatible with existing CMPs;
- To assess the impact and resolution of semantic conflicts in large-scale benchmarks and measure the effectiveness of our technique in comparison to existing CMPs;
- To highlight and address severe implications for memory management caused by the introduction of semantic conflicts into TinySTM.

2.3 Transaction Replication

Within TinySTM (and other word-based TMs) transactional statements are placed within macros delimiting the start and end points of a transaction (e.g. tmStart, tmCommit etc). These start and end points may be placed within the scope of a normal C function. Given that a transaction may require read and write access to the local variables of the encompassing function, a transaction cannot be replicated because thread cannot access the local variables of thread.

In order to allow transaction replication in TinySTM, transactions must be written to the following specifications:

- The statements of any replicable transaction must be contained within a dedicated transaction function and supplied with the necessary transactional arguments, which any thread can invoke;
- All local variables within the transaction function must be accessed via two new functions (ThdRepRead and ThdRepWrite) for reading and writing of local variable data.
and (ii) it encounters data that is session locked while executing a transaction while (i) it encounters a semantic conflict while executing a transaction attempts to resolve semantic conflicts within the context of sessions. Semantic Conflicts and Word Based STM

2.4 Sessions

Hugh2 attempts to resolve semantic conflicts within the context of a session. A non-session mode thread will enter session mode if: (i) it encounters a semantic conflict while executing a transaction and (ii) it encounters data that is session locked while executing a transaction. This ensures that the original local variables are not modified unless the transaction is committed, while allowing any thread to run the transaction with the local variable context, replicated within its write-set.

If the transaction commits, the contents of the write set are written permanently to memory, which now includes any local variable data which were added to the write set by the Thread-Replicable functions. From the perspective of the application user, completed transactions maintain the illusion that they have been executed only once and by a single thread while replication now allows any thread to execute any transaction.

2.4 Sessions

Algorithm 1: Thread Replicable Read/Write Functions.

```plaintext
function ThdRepRead(tx, addr)
if ¬tx.sessionMode then return read(addr);
entry ← hashEntry(tx, addr);
if entry.session = sessionNo(tx.id) then
  while true do
    if entry.addr = addr then
      return entry.value;
    else if entry.next = null then
      return createEntry(tx, addr, atomicRead(addr), trmask).value;
    entry ← entry.next;
else /* first time entry has been read */
  return createEntry(tx, addr, atomicRead(addr), trmask).value;

function ThdRepWrite(tx, addr, value)
if ¬tx.sessionMode then return write(addr, value);
entry ← hashEntry(tx, addr);
if entry.session = sessionNo(tx.id) then
  while true do
    if entry.addr = addr then
      updateWsetEntry(entry, value, trmask);
      return;
    else if entry.next = null then
      createEntry(tx, addr, atomicRead(addr), trmask).value;
      return;
    entry ← entry.next;
else /* first time entry has been read */
  createEntry(tx, addr, value, trmask);
```

The functions ThdRepRead and ThdRepWrite allow the concurrent reading and writing of local variables within a transactional function (see Algorithm 1). Both functions first detect whether the thread is in session mode (lines 1, 11 and 12). If so, requested read and write data is either retrieved from, entered into or updated within the write set of that thread (lines 2-10 and 13-23). This ensures that the original local variables are not modified unless the transaction is committed, while allowing any thread to run the transaction with the local variable context, replicated within its write-set.

If the transaction commits, the contents of the write set are written permanently to memory, which now includes any local variable data which were added to the write set by the Thread-Replicable functions. From the perspective of the application user, completed transactions maintain the illusion that they have been executed only once and by a single thread while replication now allows any thread to execute any transaction.

2.4 Sessions

Algorithm 2: TinySTM Handlers.

```plaintext
function onStart(tx, fn, args)
if tx.state ≠ started then return nocalltx;
if tx.nbAborts = 0 then
  setTableEntry(tx.id, fn, args);
if tx.sessionMode then
  setTableSession(tx.id, sessionCounter);
else
  return calltx;

while true do
  if (txcall ← getNextTx(tx)) = noMoreTx then
    return;
  else
    continue;
  /* my transaction wasn't committed */
  if consensusReached(sessionNo(tx.id)) then
    tx.state ← wonConsensus;
  /* I won the consensus round */
  tx.state ← started;
  invoke(txcall.ftn, txcall.args);
if onTxSuccess(tx, txcall) = 0 then
  break;

function onPreCommit(tx)
if tx.state ≠ started then return;
logEntry ← UCLogEntry(sessionNo(tx.id));
if cas(&logEntry, logEntry, tx.txMask) = fail then
  tx.state ← lostConsensus;
  rollback();
else /* I won the consensus round */
  tx.state ← wonConsensus;

function onCommit(tx)
logEntry ← UCLogEntry(sessionNo(tx.id));
if tx.state = wonConsensus then
  atomicIncrement(sessionCounter);
  tx.state ← started;
if bitIsSet(logEntry, tx.id) then
  tx.sessionMode ← false;
else /* my transaction wasn't committed */
  rollback();

function onAbort(tx, explicit)
if explicit = true then tx.sessionMode ← true;

function onTimeout(tx)
if commitCount(tx.txMask) > 0 then
  return 0;
else
  return (tx.counter ← newLimit);

function onTxSuccess(tx, txcall)
setBit(tx.txMask, txcall.id);
if decrement(tx.counter) = 0 then
  return onTimeout(tx);
return tx.counter;
```
a transaction. A session mode thread returns to non-session mode once its transaction has been committed.

In addition to the normal structures required by TinySTM, the following data structures are required to support session execution:

- A global Transaction Table is provided where the n-th entry into the table belongs to the n-th thread in the application. Threads in session mode retrieve and execute transactions stored in the table;
- A global UC is provided (a linked list) with a session counter (an integer). Each node in the list corresponds to a session and the session counter identifies the current session. Every node contains a bit mask denoting which transactions were committed for that particular session;
- Each thread possesses a flag indicating whether it is in session mode. Each thread also records its progress during a session with a state variable (which may hold the value: started, lostConsensus or wonConsensus);
- Each thread maintains a counter to record the number of times it has attempted to invoke a transaction and a bit mask to record the transactions it has executed during a session. The i-th bit of the mask corresponds to the i-th entry in the Transaction Table.

TinySTM allows custom handlers to be called upon the occurrence of several important events during the per-thread execution of a transaction. The Hugh2 CMP is mostly implemented with custom code within these handlers, specifically: onStart, onPreCommit, onCommit and onAbort. The pseudo code for these handlers is provided in Algorithm 2:

onStart contains the code that performs the iterative execution of transactions during a thread’s session mode. When a thread first executes a transaction it adds the transaction function and argument to the transaction table (lines 2 and 3). If the thread is not in session mode, then it returns from the onStart handler and executes its transaction normally (line 6). If the thread is in session mode, then the thread’s table entry is updated to hold the current value of the session counter (lines 4 and 5).

Setting the session counter value in the table essentially acts as a flag which other session mode threads can read. If their session counter values are equal, thread’s transaction can be executed as part of the current session.

Lines 7 to 15 of the onStart handler contain a while loop that performs the iterative execution of transactions. The thread first attempts to retrieve a new transaction to execute (line 8). If no more transactions are available, the thread calls an onTimeout handler (lines 31-33). If the thread has not committed any transactions, it continues reading from the table. Otherwise, the thread breaks out of the loop and returns the nocalltx constant (lines 10 and 16) indicating that the transaction should not be called after the onstart handler has returned. If consensus has been reached (lines 11-12) or the next transaction is successfully executed and no time remains (lines 14-15), the thread breaks out of the loop and returns nocalltx.

onTxSuccess is invoked when a transaction is successfully executed in session mode. The onTxSuccess handler: (i) updates the thread’s bit mask, setting the bit equal to the transaction’s index into the transaction table (line 34), and (ii) decrements the thread’s counter (line 35). If the counter has reached 0, the onTimeout handler is invoked (line 36).

onPreCommit is invoked prior to the thread attempting to reach consensus. If a thread is in session mode, then onPreCommit causes the thread to attempt to reach consensus by calling compare-and-swap (CAS) to set the status of the next entry in the UC (line 19), the thread updates its state, depending on the result of the CAS call (lines 20 and 22).

Algorithm 3: Session Lock Handlers.

```plaintext
function onSharedAccess(tx, lock)
    ctr ← sessionCounter;
    if ¬tx.sessionMode then
        if ¬sessionLocked(lock) then return proceed;
        if ctr ≠ sessionNo(lock) then return stale;
        tx.sessionMode ← true;
        return killself;
    if consensusReached(sessionNo((tx.id))) then
        return killself;
    if ¬sessionLocked(lock) then return proceed;
    if nextctr ≠ sessionNo(lock) then return stale;
    return sessionLocked;

function onLock(tx, lock, accessResult, accessType)
if ¬tx.sessionMode then
    lockval ← createTinyStmLock(lock, accessType);
    return (cas(lock.addr, lock.val, lockval) ≡ success);
if accessResult = sessionLocked then
    return true;
nextctr ← sessionCounter;
sLockValue ← createSessionLock(nextctr);
return (cas(lock.addr, lock.val, sLockVal) = success);
```

onCommit is called after the onPreCommit handler has been invoked. If the calling thread is in session mode and it decided the consensus result, then it atomically increments the session counter (line 25). This will indicate to other threads that the session has terminated. In line 27, threads check the UC to determine whether their transaction was committed. If the thread’s transaction is committed, then the thread leaves session mode (line 28), otherwise the thread rolls-back execution and will attempt a new session (line 29).

onAbort is invoked whenever a transaction aborts. A flag is supplied to the abort handler to identify whether the abort was made implicitly (a concurrent conflict) or explicitly (a semantic conflict). In the case of an explicit abort, this acts as a signal to the aborting thread that a semantic conflict has been encountered and so the thread sets an internal flag to show that it has entered session mode.

Session Locks As with conventional TinySTM, reading and writing of shared data is preceded by the locking of such data (when TinySTM is configured to use eager locking and visible reads). To this end, TinySTM provides both read and write locks, but to accommodate our CMP we have added an extra type of lock called a session lock. A session lock differs from a read/write lock in two fundamental ways:

- Once locked, a session lock grants access to a word of shared data for any thread operating in the same session, hence a session lock is locked only once per session;
- A session lock is never explicitly unlocked. Rather session lock have a viable life-time for the duration of the session in which they were initiated. Once the session has ended, the session lock is considered stale and may be removed at the discretion of any encountering thread.
The properties of the session lock are particularly conducive to the pattern of execution within a session. Figure 2 exemplifies the benefits of using the session lock. Observe that, threads 1 engages session lock on data words a, b and c but gains immediate access to data item d because thread 1 is able to share the session lock (likewise, thread 2 does not have to lock data items c and d). Threads will execute the same set of transactions, albeit in different permutations, within a session. Note that because session locks are shared, repeated executions of a transaction do not have to acquire any locks to proceed.

In TinySTM, a lock is represented by a word-sized integer, with the value of the last two bits denoting the type of lock (binary 0 is unlocked, 1 denotes write locked and 2 denotes read locked). A session lock is represented by setting both bits (i.e. binary value 3). The remaining bits of the word value hold the session number in which the lock was set. Algorithm 3 shows two handlers which are invoked when dealing with session locks:

- **onSharedAccess** is called before a shared word is locked for reading or writing. Non-session mode threads can proceed to content the shared data (line 3) if it is not session locked or if the session lock is stale (line 4). Otherwise the thread must abort its transaction (line 6), after entering session mode (line 5). Conversely, session mode threads can access session locked data (line 9), or install a new session lock if an existing lock is stale (line 8). If shared data is read or write locked, session mode threads attempt to abort the locking thread and install a session lock (session mode threads take precedence over non-session mode threads).

- **onLock** is called whenever a thread attempts to lock shared data. Non-session mode threads create a normal TinySTM type lock and attempt to lock the data. Threads in session mode can immediately access session locked data because they can share the lock. Otherwise, session mode threads attempt to session lock the data (line 18).

In addition to TinySTM locks, session locks are installed using CAS to ensure that the status of the shared data has not changed between reading the lock value and the subsequent attempt to lock the shared data.

### Figure 2. Hypothetical time-line showing the acquisition and release of session locks by various threads when using the Hugh2 CMP.

3. Related Work

Replication has been previously explored in the context of Transactional Memory to provide fault-tolerance and service-availability across multiple hosts. Since Active Replication (AR) was introduced to distributed systems [24], transaction replication has been explored, principally, in the context of Distributed Software Transactional Memory (DSTM), based upon an Atomic Broadcast (AB) service [6]. AB was enhanced in [17], to provide optimistic operations to allow greater parallel execution based on static access patterns of transactions. Palmieri et al have provided advancements by utilising transaction replication in AGGRO [20] and speculative execution in OSARE [21] to accommodate realistic transactions with unpredictable access patterns.

In comparison to the cited research in DSTM, where replication provides fault-tolerance and service availability, Hugh2 provides transaction replication to allow a greater number of threads to participate in the exploration for a suitable transaction execution schedule. To be precise, Hugh2 seeks a schedule of execution wherein the occurrence of semantic conflicts is minimised, and replication allows multiple threads to participate in that endeavour. In theory, as greater parallel processing power is afforded by the host platform, replication should allow greater parallel exploration and enhance the performance of our approach.

A range of CMPs exist which relate to Hugh2. These can be categorised under two general schemes: wait based and schedule-based. Early CMPs can be regarded as wait-based [12, 23]. The benefit of such wait based approaches (Greedy, Karma, Polka etc), is that they have shown to be trivial to implement and versatile while offering good performance improvements to contention management in STM. Heber et al, however, noted in [14] an inherent inefficiency with wait-based approaches due to the difficulty in finding an adequate back-off period, given the highly dynamic nature of transaction execution in STMs.

An alternative to wait-based CMPs are several relatively newer schedule-based CMPs which typically reschedule aborted transactions and/or perform serialisation of transactions by executing contentious transactions with a single thread. Bai et al [2] provides an early example of such an approach. Bai et al produced several “transaction executor” models, where threads take on the roles of ‘producer’, ‘executor’ or ‘worker’. The aim of these models is to
equitably distribute transactions as ‘jobs’ among the threads of an application. ‘Keys’ are then used to predict the likelihood that conflicts will arise between executing transactions. When transactions are likely to conflict, they are scheduled to be executed by the same ‘worker’ thread, thus enforcing serialisation.

CAR-STM [8] and Steal on Abort [1] are additional schedule-based CMPs which follow a similar task executor approach, where transactional jobs are assigned to per-thread work queues. Both CAR-STM and Steal on Abort move aborted transactions to the work queues of conflicting transactions upon the occurrence of a conflict, to serialise the conflicting transaction’s execution. CAR-STM also predicts the likelihood of transactional conflicts to prevent conflicts from arising. Steal on Abort experiments with various techniques when moving transactions to the work queues of other threads (a transaction can be placed at the end or the start of a work queue). With Steal on Abort, additional work queues can be created when the number of transactional jobs increases.

Hugh2 differs from the cited approaches of both wait-based and schedule-based CMPs, insofar as Hugh2 is the only approach which focuses on the resolution of semantic conflicts. In addition, Hugh2 requires a single transaction table to hold transactional jobs, but does not require the overhead of a thread pool to administer such jobs. Hugh2 also uses replication to explore multiple schedules in parallel during the process of contention management.

In comparison to Hugh2, several approaches to STM have been developed which rely on a Universal Construction (UC). The initial work on UC was presented by Herlihy [15] to enable multiple threads to access shared data structures via a wait-free algorithm. Wamhoff [25] and Chuong [4] combined the UC technique with transactions to handle certain failure conditions. Crain et al have shown that it is possible to remove the abort semantics of STM using a UC [5]. While the cited approaches apply the UC technique for a STM system, Hugh2 differs insofar as a UC is used for contention management.

**Internal versus External Transaction Reordering**

TL-STM [3] is an adaptation of SwissTm which incorporates Thread-Level-Speculation (TLS) into memory transactions. Principally, TL-STM bears similarity to Hugh2 insofar that the parallelism of the host platform is used to explore different permutations of transactional elements while resolving concurrency conflicts. More specifically, TL-STM seeks to enhance transactional throughput by reordering the internal execution elements of a transaction to better reflect concurrent schedules of execution. Conversely, Hugh2 seeks to reorder or schedule transactions to accommodate semantic schedules of execution. TL-STM applies internal reordering based on the semantics of a transaction, while Hugh2 applies external reordering based on the semantics of an application.

### 4. Evaluation

In this section we present results from a series of benchmarks to demonstrate the performance of our system. The tests were carried out on a desktop PC featuring 2 x dual-core 3.07GHz Intel(R) processors (i3) with 4GB of RAM. The Operating System used was Ubuntu (Linux) version 13.04 and the Transactional Memory software was TinySTM version 1.04 with the Write-Back, Eager Transactional Locking scheme using visible reads.

Experiments were carried out with increasing numbers of threads (from 2 to 16) with each run executed 5 times with the average results provided. Two existing CMPs were used as a measure of comparison with Hugh2, specifically Karma and Polka [23].

Two benchmarks were used to test the performance of Hugh2. The first scenario (bank) is provided in the TinySTM software and allows the execution of a number of transaction types on a hypothetic set of bank accounts. The ‘bank’ in this case is an array of account data structures. The second scenario (vacation) is part of the STAMP benchmark suite [19] and provides transactional accesses over several red-black trees to represent a holiday booking database system. Both scenarios provide update, read-all and write-all transaction types which can be generated at varying intensities.

Transactions from the vacation scenario differ from the bank simulation insofar as they tend to execute more statements of greater complexity.

To test the effectiveness of our system, semantic transactions were introduced into bank and vacation. In the bank scenario, two extra transactions (called service charge and pay interest) were created which explicitly call abort based on the balance of certain bank accounts. In the vacation scenario an additional red-black tree was created and two transaction types (called create customer and remove customer) which add and remove nodes while explicitly aborting if the contents of the tree is deemed incorrect.

The semantic transactions essentially allow us to introduce a consumer/producer pattern into the scenario, such that a consumer transaction will fail if it is not preceded by a producer (and visa versa). The semantic transactions interact with numerous other shared data elements, so it is expected that if semantic transactions must abort frequently, their activity will be detrimental to the success rate of the remaining transactions in the scenario. By increasing the number of semantic transaction threads in a scenario (as a proportion of the total number of threads), we can effectively measure the impact of semantic conflicts within the scenarios (for example, we might set up a scenario with 16 threads and specify that 8 of the threads execute semantic transactions to observe the effects of 50% semantic conflicts on the throughput of the application).

#### 4.1 Semantic Conflicts and Memory

**TinySTM** uses an epoch based garbage collection algorithm to handle the disposal of transaction write-sets and other memory allocated within a transaction. Because TinySTM allows any thread to access memory locations containing the write-set of another thread, the garbage collector must ensure that (write-set) memory is not freed until all threads have advanced sufficiently to a point where they will not access old write-set memory.

Should a transaction have to abort for any reason, then a new write-set is allocated from memory and the old write-set is passed to the garbage collection algorithm. If a situation should arise where a large number of transactions are aborted in a short period of time, then the TinySTM garbage collector must retain a considerable amount of memory until old write-set data is released. Furthermore, TinySTM requires that write-set memory be allocated such that the cache line of the new memory pointer is an address with the last-die the disposal of transaction write-sets and other memory allocated within a transaction. Because TinySTM allows any thread to access memory locations containing the write-set of another thread, the garbage collector must ensure that (write-set) memory is not freed until all threads have advanced sufficiently to a point where they will not access old write-set memory.

Should a transaction have to abort for any reason, then a new write-set is allocated from memory and the old write-set is passed to the garbage collection algorithm. If a situation should arise where a large number of transactions are aborted in a short period of time, then the TinySTM garbage collector must retain a considerable amount of memory until old write-set data is released. Furthermore, TinySTM requires that write-set memory be allocated such that the cache line of the new memory pointer is an address with the last-die the disposal of transaction write-sets and other memory allocated within a transaction. Because TinySTM allows any thread to access memory locations containing the write-set of another thread, the garbage collector must ensure that (write-set) memory is not freed until all threads have advanced sufficiently to a point where they will not access old write-set memory.

Should a transaction have to abort for any reason, then a new write-set is allocated from memory and the old write-set is passed to the garbage collection algorithm. If a situation should arise where a large number of transactions are aborted in a short period of time, then the TinySTM garbage collector must retain a considerable amount of memory until old write-set data is released. Furthermore, TinySTM requires that write-set memory be allocated such that the cache line of the new memory pointer is an address with the last-die the disposal of transaction write-sets and other memory allocated within a transaction. Because TinySTM allows any thread to access memory locations containing the write-set of another thread, the garbage collector must ensure that (write-set) memory is not freed until all threads have advanced sufficiently to a point where they will not access old write-set memory.
To address this critical problem, we enhanced the garbage collection algorithm so that threads pause individual execution and wait until their memory consumption falls below a threshold before executing another transaction. Initial testing began with several executions of the scenarios to determine the required platform specific thresholds to prevent the application failing due to lack of memory. The thresholds were then supplied to the garbage collection algorithm. Once a thread exceeds the threshold, it repeatedly reads the value of the global counter, until the global counter has been incremented (by some other thread committing a transaction). The waiting thread may then attempt to clean up memory, which will eventually bring the amount of stored memory back below the threshold so that the thread can continue executing more transactions.

The behaviour of the garbage collector is clearly of considerable importance if it can cause the application to fail because of exceeded memory. Although our solution prevents this from happening, causing threads to wait is not an approach we regard as a long-term solution, given that the waiting makes it harder to judge the effectiveness of our CMP. For greater clarification of the results, we have therefore measured the effects of waiting on the scenario. Figure 7 contains provide several graphs showing the average number of iterations threads made when attempting to clean up memory, and in Section 4.4 we describe these observations in more detail.

4.2 Transaction Throughput

Figure 3 provides graphs showing different results for the bank scenario. Y-axes show the number of transactions committed per second and X-axes show the number of threads used. Graph 3(A) provides comparison between the Karma, Polka and Hugh2 CMPs in the absence of semantic conflicts. Note that the system which employs Hugh2 for semantic conflicts, resorts to calling the Karma on occurrence of concurrent conflicts. As there are no semantic conflicts being generated in graph 3(A), the performance of Hugh2 and Karma are practically the same (as one would expect).

In Graph 3(B) semantic conflicts have been introduced into the scenario at a rate where 50% of the threads in the scenario generate semantic transactions in the case of thread numbers: 4, 8 and 16. At this point the throughput for Karma and Polka have both fallen noticeably relative to the throughput for Hugh2 which has increased substantially. In Graph 3(C) semantic transactions are being created by 100% of the threads in the scenario. Once again the throughput for both Karma and Polka has been reduced dramatically, whereas Hugh2 outperforms both CMPs significantly.

In Figure 4 we can see the throughput results for the vacation scenario. The results are presented in the same format as the bank scenario. Once again, in Graph 4(A) no semantic conflicts are created and the performance of Hugh2 approximately mirrors that of Karma. In Graph 4(B), 50% of the threads in the scenario perform semantic transactions in the case of thread numbers 4, 8 and 16. Once again, the throughput of Karma and Polka diminishes whereas the throughput for Hugh2 has increased. This pattern is replicated in Graph 4(C) where 100% of the threads generate semantic conflicts.

In terms of throughput, when comparing the vacation scenario to the bank scenario we can see that the Polka CMP mostly outperforms both the Karma and Hugh2 CMPs when semantic conflicts are absent. This is unsurprising as Polka enhances the Karma CMP and has been cited as providing the best average performance of wait-based CMPs [1] (one notable exception, however, is in the vacation scenario when 16 threads are used). More significantly however, it is encouraging to see that Hugh2 can function in combina-

1 Two or more threads are required to resolve semantic conflicts (i.e. a producer and consumer). To show 50% semantic conflicts therefore requires at least four or more threads. The results for 2 threads show 0% semantic conflicts instead.
tion with an existing CMP (in this case Karma), without degrading the performance with respect to resolving concurrent conflicts. Conversely, observe that as semantic conflicts are introduced, neither Karma or Polka can approach the effectiveness of Hugh2 in maintaining a higher level of transaction throughput. Although Polka almost always produces a higher throughput than Karma, neither approach produces good performance when semantic conflicts are present, regardless of scenario. One may observe however that throughput diminishes for Hugh2, in the case of 50% semantic conflicts. This diminishing performance is witnessed to a lesser extent with 100% semantic conflicts, suggesting that the greater occurrence of threads producing concurrent conflicts has a negative impact on Hugh2.

### 4.3 Maximum Transaction Retries

Figures 5 and 6 present results showing the average maximum retries for a transaction during the bank and vacation scenarios respectively. The format of the graphs in Figures 5 and 6, mirrors the previous results for transaction throughput, as described in Section 4.2. The exception is that the Y-axis now shows the average retries instead of transaction throughput.

A higher the average maximum number of retries, is indicative of threads experiencing significant difficulty in resolving a semantic conflict. If the average maximum is high, then is suggests semantic conflicts were particularly contentious in the scenario. It is therefore expected that the average maximum retries will increase as the number of semantic conflicts increase for the Polka and Karma managers, whereas this should not be the case for the Hugh2 CMP.

Graph 5(A) provides comparison between the Karma, Polka and Hugh2 CMPs where there are no semantic conflicts introduced into the scenario. The Polka CMP produces the smallest average maximum retries, which is unsurprising given that the graphs in Figure 3 established that Polka produced the highest throughput when no semantic conflicts were present.

In Graph 5(B), semantic conflicts have been introduced into the scenario at a rate where 50% of the threads in the scenario generate semantic transactions in the case of thread numbers: 4, 8 and 16. A substantial increase in average maximum transaction retries is observable in all CMPs, however, the Hugh2 CMP performance is the best. In addition, the improved performance of the Polka CMP in comparison to the Karma CMP has diminished once semantic conflicts are introduced. This would suggest that neither policy is more effective than the other at resolving semantic conflicts.

In Graph 5(C) semantic transactions are being created by 100% of the threads in the scenario. Once again the average maximum number of retries has increased for both Karma and Polka CMPs. In the case of Hugh2, the average maximum has fallen. Once again neither Karma or Polka tackles semantic conflicts more effectively than Hugh2.

Figure 6 shows the average maximum retries for the vacation scenario. Graph 6(A) compares Hugh2, Karma and Polka when no semantic conflicts are present. Graphs 6(B) and 6(C) show the same comparisons with 50% and 100% semantic conflicts, respectively. In the vacation scenario, the average maximum retries: (i) increases for both Karma and Hugh2 when no semantic conflicts are present (graph 6(A)) and (ii) decreases for Hugh2, when semantic conflicts are present (graphs 6(B) and 6(C)).

With the vacation scenario producing longer ‘non-semantic’ transactions than bank, these results are expected because extending the duration of a transaction incurs a higher likelihood that a conflict will occur and therefore the chances of retrying a transaction are higher. Conversely, when the shorter semantic transaction dominate the scenario, the main determinant of performance becomes the resolution of semantic conflicts (where Hugh2 has the advantage).
when a transaction aborts, the thread must allocate a new write-lock. The demands for memory are the highest given that for shared data is greatest and so the frequency of transaction aborts of 16 threads. With 16 threads this is expected because contention and 6, for instance, show the highest number of retries in the case parallel resources during the scenario. The graphs in Figures 5 and 6, for instance, show the highest number of retries in the case of 16 threads. With 16 threads this is expected because contention for shared data is greatest and so the frequency of transaction aborts is the highest. The demands for memory are the highest given that when a transaction aborts, the thread must allocate a new write-set from memory. Combining the memory being demanded by the high frequency of transaction aborts and the relatively low number of processors on the platform, it is not surprising that occasionally, demands for memory exceed the set thresholds, causing threads to hold off allocating more memory.

In Graph 7(B), semantic conflicts have been introduced into the scenario at a rate of 50%. The average number of iterations has increased substantially (between 100,000 and 1,000,000) for the Karma and Polka CMPs. Waiting now occurs in the case of 4, 8 and 16 threads. In the case of Hugh2, waiting can also be observed but only in the case of 8 and 16 threads. The average number of iterations is also substantially lower than with Karma and Polka (approximately 1,000 to 10,000). As Hugh2 resolves the semantic conflicts, this reduces the frequency of transaction aborts and hence the demands for new memory are significantly lower.

In Graph 7(C), all threads in the scenario generate semantic conflicts. We can observe that as each X-axis has produced a value, waiting occurred now regardless of the number of threads in the case of Karma and Polka (although the average waiting time has not increased significantly). Conversely, there is no significant change in behaviour in the case of Hugh2 when the percentage of semantic transactions is increased; the average waiting time has not increased significantly and waiting still only occurs in the case of 8 and 16 threads.

The results of Figure 7 seem to suggest that: (i) Hugh2 can more effectively deal with semantic conflicts and (ii) this has a observably positive impact on the memory consumption of threads in the scenario. It is interesting to note that average waiting decreased as more threads were added to the scenario in the case of Karma and Polka. A likely reason for this is that the addition of more threads helped increase the changes that some random transaction execution schedule would occur which would resolve a semantic conflict in the case of Karma and Polka. Note however that comparable performance of both Karma and Polka in terms of average waiting times; it seems that neither the strategy of Polka or Karma had a noticeable impact on resolving semantic conflicts.

4.4 Garbage Collection Performance

Figure 7 illustrates the effects of semantic conflicts on the operation of the garbage collector (GC) used in TinySTM. Note that due to limited space, we have only provided the results from the vacation scenario (although the implications raised by the vacation scenario apply equally to the bank scenario). As described in Section 4.1, we adapted the GC to cause threads to wait until their memory consumption had fallen below a pre-determined threshold. The graphs presented in Figure 7 show the extent of thread waiting, providing the average number of iterations threads performed (once their memory consumption exceeded the threshold) until another thread had progressed, thus allowing memory to be reclaimed.

Graph 7(A) shows the average number of iterations performed when no semantic conflicts are generated. In the cases where 2, 4 and 8 threads were used, observe that the number of iterations was 0 for all CMPs. This is expected because concurrent conflicts alone should still produce liveness of the global clock, which in turn is necessary to prevent the GC from accumulating too much memory. Observe, however, that when 16 threads were used, the threshold was exceeded in the case of all CMPs, and the average number of iterations was approximately 1000 under each CMP, (although Polka produced the lowest).

The reason why the threshold was exceeded in the case of 16 threads may be a result of high contention among threads for parallel resources during the scenario. The graphs in Figures 5 and 6, for instance, show the highest number of retries in the case of 16 threads. With 16 threads this is expected because contention for shared data is greatest and so the frequency of transaction aborts is the highest. The demands for memory are the highest given that when a transaction aborts, the thread must allocate a new write-set from memory. Combining the memory being demanded by the high frequency of transaction aborts and the relatively low number of processors on the platform, it is not surprising that occasionally, demands for memory exceed the set thresholds, causing threads to hold off allocating more memory.

In Graph 7(B), semantic conflicts have been introduced into the scenario at a rate of 50%. The average number of iterations has increased substantially (between 100,000 and 1,000,000) for the Karma and Polka CMPs. Waiting now occurs in the case of 4, 8 and 16 threads. In the case of Hugh2, waiting can also be observed but only in the case of 8 and 16 threads. The average number of iterations is also substantially lower than with Karma and Polka (approximately 1,000 to 10,000). As Hugh2 resolves the semantic conflicts, this reduces the frequency of transaction aborts and hence the demands for new memory are significantly lower.

In Graph 7(C), all threads in the scenario generate semantic conflicts. We can observe that as each X-axis has produced a value, waiting occurred now regardless of the number of threads in the case of Karma and Polka (although the average waiting time has not increased significantly). Conversely, there is no significant change in behaviour in the case of Hugh2 when the percentage of semantic transactions is increased; the average waiting time has not increased significantly and waiting still only occurs in the case of 8 and 16 threads.

The results of Figure 7 seem to suggest that: (i) Hugh2 can more effectively deal with semantic conflicts and (ii) this has a observably positive impact on the memory consumption of threads in the scenario. It is interesting to note that average waiting decreased as more threads were added to the scenario in the case of Karma and Polka. A likely reason for this is that the addition of more threads helped increase the chances that some random transaction execution schedule would occur which would resolve a semantic conflict in the case of Karma and Polka. Note however that comparable performance of both Karma and Polka in terms of average waiting times; it seems that neither the strategy of Polka or Karma had a noticeable impact on resolving semantic conflicts.

5. Conclusion

This paper presents Hugh2, a CMP which deals with semantic conflicts via the replication and speculative execution of aborted transactions. We have described how Hugh2 can be integrated within a word-based STM using a new session locking mechanism. Two sophisticated benchmarks were then adapted to generate semantic conflicts and we demonstrated performance improvements, including transaction throughput, once semantic conflicts are introduced.

The evaluation of our approach suggests that STM performance can be severely affected by semantic conflicts when combined with an epoch based garbage collection (as used in TinySTM). In future we will seek to address the short-comings of the garbage collector, to ensure memory is more effectively managed when semantic conflicts increase. Given that Hugh2 can be incorporated with any existing CMP, it would be interesting to test the performance of Hugh2 against a wider range of CMPs. In addition, incorporating semantic conflicts into the remaining STAMP benchmarks may be useful in order to observe how semantic conflicts affect a diverse range of scenarios.

We believe the session lock mechanism raises some exciting possibilities for adapting our work with a distributed STM application. In particular, given that session locks do not need to be explicitly unlocked, this may provide a greater degree of scalability in the context of DSTM. While existing DSTM generally use replication for the purposes of availability and fault-tolerance, we have applied replication as a means of allowing multiple threads to speculatively explore a range of transaction schedules in order to reduce conflicts of a semantic nature.
References


