Extending Contention Managers for User-Defined Priority-Based Transactions

Justin Gottschlich  Daniel A. Connors
Department of Electrical and Computer Engineering
University of Colorado at Boulder
{gottschl, dconnors}@colorado.edu

Abstract
Transactional memory (TM) contention management (CM) is the process of handling memory conflicts in transactions. Contention managers were first proposed as a way of separating transactional progress from correctness and to prevent transactional starvation. While significant work in the area of contention management has been done, most prior work has focused on preventing starvation through various fairness policies. Real-time systems, or systems that have strict requirements for task behavior, usually guarantee such behavior through priority scheduling. As such, these systems require more than the current CM goal of starvation prevention, they require partial task ordering guarantees specified by the user.

This work considers the concrete implementation and implications of TM contention manager extension for user-defined priority-based transactions as specifically identified as consequent by prior contention management research. Our primary focus is centered on the correlation between priority scheduling and consistency checking. We demonstrate how transactional schedulers built into contention managers handle user-defined priority-based transactions and the limitations of the scheduler due to the consistency checking model.

1. Introduction
Parallel computers are the industry standard [1, 2, 4, 5] and, as such, programmers are writing more parallel programs [19, 30]. However, traditional parallel synchronization primitives, such as, locks, monitors and semaphores are exceptionally difficult to program correctly [3, 4, 17, 18]. These primitives exhibit nondeterministic behavior in parallel code which give rise to a number of problems not found in non-parallel code, like, deadlocks, livelocks, lock convoys and priority inversion [10, 17, 21]. A recently developed synchronization concept, called transactional memory (TM), shows promise in overcoming some conventional parallel problems and simplifying the task of writing correct, parallel code. Figure 1 illustrates how DracoSTM, a C++ lock-based software transactional memory (STM) system, sets a globally shared integer.

```
native_trans<int> global_int;
void set_global(int val)
{
  for (;;)
  {
    try
    {
      transaction t;
      t.write(global_int).value() = val;
      t.end_transaction();
      break;
    }
    catch (aborted_transaction_exception&) {} 
  }
}
```

Figure 1. DracoSTM transaction setting a globally shared integer.

Transactional memory simplifies parallel programming by breaking parallel tasks into transactions (as shown in Figure 1). These transactions exhibit database-like atomicity, consistency and isolation [28], but not durability and are proving to be significantly relevant to all computer systems as argued by Moss and Rajwar [29]. There are two primary types of TM systems: lock-based systems, which use locking primitives at their core (e.g. mutex locks) [6, 7, 9, 12] and non-blocking systems, which use non-blocking atomic primitives at their core (e.g. compare-and-swap (CAS)) [15, 16, 21, 27]. Recently, systems are emerging which implement both non-blocking and lock-based aspects, such as RSTM v3 [33], to gain benefits of both types of systems. The specific implementation of the TM system is what enables (or disables) it to overcome nondeterministic problems [8, 26]. The concentration of this work is centered on the connection between a TM system’s consistency checking model and its ability to handle user-defined priority-based transactions.

Priority-Based Tasks. Real-time systems or systems that have strict requirements for task behavior, such as deadline-drive systems, usually guarantee such behavior through priority scheduling [25]. While significant prior TM contention management (CM) research has been done, such as the work of Guerraoui et al. [13, 14] and Scherer and Scott [22, 23], its attention has been primarily focused on preventing starvation through fairness. In many systems preventing starvation may be sufficient, yet some systems (e.g. deadline-driven or real-time systems) require stronger behavioral guarantees. In these cases, user-defined priority-based transactions are necessary.

Approach. This work extends the prior TM contention management research of Guerraoui et al. and Scherer and Scott by concretely implementing user-defined contention managers as determined consequential by Guerraoui et al. [13]. We approach this by first presenting a brief background of TM aspects important to understanding the complexities of contention management. Next, we review and expand upon prior contention management work. We
then show how consistency models play a significant role in the correctness and capability of priority-based transactional scheduling. We then build user-defined priority-based transactions with DracoSTM, our C++ lock-based STM library, demonstrating how contention management frameworks work with different consistency checking models. Last, we present our experimental results.

2. Background
Throughout this work we use the taxonomy presented by Larus and Rajwar in their Transactional Memory text [24] to assist in promoting a single vocabulary for transactional memory. Some of these basic concepts (as well as some others) are briefly explained below.

2.1 Basic Concepts

Attacking & Victim Transactions. Following the terminology of Guerraoui et. al. we refer to transactions which can identify a memory conflict in another in-flight transaction as attacking transactions. Transactions which are targeted by attacking transactions are referred to as victim transactions [13]. The attacking and victim transaction terms help simplify the invalidation process.

An example of attacking and victim transactions is as follows.
Consider transaction $T_v$, a victim transaction and transaction $T_a$, an attacking transaction. $T_v$ writes to memory location $L_0$. $T_a$ then tries to write to $L_0$. If our STM system only allows single-writers, both $T_a$ and $T_v$ cannot write to $L_0$ at the same time. As $T_a$ is the second transaction attempting to write to $L_0$, the single-writer semantics require it to deal with the conflict located at $L_0$. Handling this conflict makes $T_a$ the attacking transaction as it decides if $T_a$ is aborted. $T_v$ is the victim transaction since $T_a$ may abort it.

Eager & Lazy Acquire. When transactions read and write to memory, they usually do so in different ways. For transaction $T_1$ to read memory location $L_0$, it can simply add some type of reference to itself (or at the memory location $L_0$) to indicate it is being read. Usually TM systems allow multiple readers of the same memory location. As such, any number of transactional readers may coexist for location $L_0$.

Written memory behaves in a different manner. Written memory must be exclusively acquired by a transaction at some point in the transaction’s lifetime. Exclusive access to written memory is required because multiple writers cannot simultaneously update the same piece of memory without introducing an inconsistency. To explain this, consider transaction $T_1$ and $T_2$ both attempting to increment the integer stored at location $L_0$. $L_0$’s initial value is 0. $T_1$ reads $L_0$ and increments it. Then $T_2$ reads $L_0$ and increments it. Both transactions read the value 0 and store the value 1. When both transactions commit they both write the value 1 at location $L_0$. However, the correct result for both transactions incrementing location $L_0$ is 2. As such, exclusive memory access must be obtained by a transaction before it can commit writes. The primary ways to acquire exclusive memory access is eagerly or lazily.

Eagerly acquired memory is obtained when the transaction performs its write operation. Lazily acquired memory is obtained after the write operation (usually at commit-time). Each memory acquisition type has different benefits. Some early STM systems only allowed single-writers and as such always performed eager memory acquisition. Other systems have begun allowing lazy memory acquisition as multiple-writers can result in different performance benefits. Some systems, such as FSTM, RSTM and DracoSTM allow for both eager and lazy acquires.

Eager acquires can help performance by preventing wasted operations since two transactions writing to the same memory cannot both commit. However, an attacking eager acquire transaction may abort a conflicting victim transaction only to be aborted later by another transaction. Lazy acquires can help performance by allowing the fastest transaction (the first transaction to complete its workload) to commit and overcomes the eager acquire scenario of cascading aborts. Lazy acquire also minimizes false identification of reader conflicts by delaying conflict identification giving the reader transactions a chance to complete first and thus result in no conflict at all. However, lazy acquire delays the notification of a doomed transaction which can result in wasted work for all conflicting transactions.

Visible & Invisible Readers. When a transaction stores its read memory so other transactions can see it, it is called a visible reader. When a transaction stores its read memory so other transactions cannot see it, it is called an invisible reader. Visible and invisible readers are directly related to eager and lazy acquires. If an attacking transaction using eager or lazy acquire has visible readers, it can identify both write-write conflicts and write-read conflicts at the time of acquisition. If an attacking transaction has invisible readers, it can only identify write-write conflicts. Write-read conflicts must be identified by the invisible readers later.

The principal difference in visible and invisible readers is in the consistency checking model. Visible readers allow attacking transactions to invalidate write-read conflicts, while invisible readers require the reading transactions to validate themselves.

2.2 Consistency Checking
The process of finding a memory conflict is called consistency checking. Two high-level types of consistency checking exist: validation and invalidation.

Validation. When a TM system performs validation, each transaction checks its own read and write sets for conflicts against global memory. If a conflict is found, the transaction (usually) must abort itself. This is due to validating conflicts found in committed global memory which are incapable of being undone. Systems based entirely on validation employ invisible readers. Each transaction which reads an object does so in an invisible way, so other transactions cannot see these reads. When a transaction acquires an object for writing, it does not invalidate any of the reading transactions. The reader transactions must validate themselves at some point after the writer transaction has acquired the object.

Invalidation. When a TM system performs invalidation, each transaction checks its own write set for conflicts against other in-flight transactions. If a conflict is found the invalidating (attacking) transaction can either flag the in-flight (victim) transaction as invalid, wait or abort itself. Fully invalidating systems require that all transactions have visible readers, which allow the attacking transaction to identify and flag write-write conflicts and write-read conflicts. Some partial invalidating systems employ invisible readers which allow attacking transactions to identify write-write conflicts but not write-read conflicts. In such partially invalidating systems, write-read conflicts are validated by readers at a later time.

Some systems perform both validation and invalidation, such as RSTM [27]. RSTM does this by containing a limited number of visible readers which if exceeded results in the remaining readers being invisible. For example, when RSTM performs eager invalidation a transaction that writes to memory does so eagerly. The attacking transaction then invalidates any other existing write conflicts for the write memory as well as read conflicts through the object’s visible reader list. If other transactions are invisibly reading the memory, those transactions must perform validation on themselves at a later time. If RSTM is performing lazy invalidation a similar process occurs except memory is acquired in a lazy fashion, rather than an eager one. In both cases RSTM may perform invalidation and validation based on the number of readers.
User-Defined Priority-Based Transactions. Validation and invalidation consistency checking are both practical solutions to transactional memory for different classes of problems. When in-flight transactions are low and memory usage is high, invalidation may be preferred [11]. When transactional traffic is high and transactions are small, validation may be preferred. Neither type of consistency checking seems to perform universally better than the other. Though certain classes of problems perform better under different consistency checking models, the ramifications of these consistency checking schemes performing within strict user-defined priority-based transactional systems is unclear from prior research. Our work clarifies the advantages and disadvantages of validation and invalidation in user-defined priority-based transactional environments.

3. Related Work

Over the last five years, a solid foundation of work has been performed on contention managers. The two primary contention management works we examine and extend are Scherer and Scott’s [22, 23] and Guerraoui et al. [13, 14]. Scherer’s CM system was implemented in DSTM (a Java-based system created by Herlihy et al. [20]) while Guerraoui’s was implemented in SXM (a C#-based system).

3.1 Contention Management: Scherer and Scott

Scherer and Scott first proposed their contention management schemes using the Dynamic Software Transactional Memory (DSTM) of Herlihy et al. [20]. The original version of DSTM supported single-writers only. Single-writer TM systems only allow a single transaction to write to a specific piece of memory, even if this writing is done off to the side, as is the case in DSTM’s design. The following encapsulates a variety of their findings as well as more recent observations from Larus and Rajwar [24], and Scott [32].

Invisible Readers & Eager Write-Write Invalidation. Single-writer TM designs require eager acquire semantics. When an attacking transaction attempts to write to a piece of memory previously written to by another victim transaction, the system must immediately handle the memory contention. Therefore, single-writer TM systems must employ, at least, partial invalidation, as the attacking transaction identifies a conflict with a victim transaction and some action must be taken as the TM system only supports one writer. The attacking transaction can abort the victim transaction, wait or abort itself [24, 32]. However, any action taken eagerly is not guaranteed to be correct. If the attacking transaction aborts the victim transaction, it may later be aborted itself by another transaction from a different conflict. If the attacking transaction allowed the victim transaction to continue, the victim transaction may have committed successfully. If the attacking transaction aborts itself, the same scenario as described above could occur, but in reverse. Finally, if the attacking transaction stalls, a deadlock can occur. There is no clear correct choice for a contention manager to make when eager write-write conflicts are found.

Visible Readers & Eager Write-Read Invalidation. If readers are visible and the attacking transaction also invalidates them, the system is eager invalidating [32]. The original DSTM design only supported invisible readers, but was later extended to support visible readers so it could perform eager invalidation. While eager invalidation can manage write-read conflicts as well as write-write conflicts, it does so on a speculative basis. Since neither the writer nor reader transaction are in their commit phase, it is not yet clear if the conflict is true. If the reader transaction commits prior to the writer transaction, the conflict is false as the reader transaction’s data is consistent. Therefore, if an eager invalidation aborts either the reader transaction or the writer transaction there is no way of ensuring any such abort is necessary.

Visible Readers & Commit-Time Invalidation. Complete commit-time invalidation performs both write-write and write-read conflict detection by a committing transaction. When a conflict is found by a transaction in the commit phase of a commit-time invalidating system the transaction must either: back out of the commit phase and stall, abort the conflicting victim transaction or abort itself. Systems which implement transactions that back out of the commit phase and stall are dangerous as they can lead to deadlock. If the attacking transaction chooses to commit, all write-write and write-read conflicts the committing transaction finds must be aborted as they are true conflicts. If the attacking transaction chooses to abort itself, none of the conflicts found by the attacking transaction should be aborted.

Invisible Readers. As noted by Scherer and Scott, if readers are invisible there is no hook through which contention management can address the potential conflict [22]. In short, a CM cannot perform any conflict resolution on write-write or write-read conflicts in validating systems since the attacking writer transaction does not see the conflicting writer or reader transactions while they are still in-flight.

Incremental Validation & Commit-Time Validation. Both incremental and commit-time validation use invisible readers. Incremental validation performs iterative validation each time a new memory location is opened for reading or writing. Commit-time validation is slightly different than incremental validation in that validation is only performed once, at commit-time. Unlike invalidation, validating systems only locate conflicts which have already been committed. When such a conflict is found, a validating transaction (usually) must abort itself. Incremental validation may be preferred over commit-time validation due to its early notification of committed conflicts. From this view, incremental validation can improve performance by earlier doomed transaction notification. However, commit-time validation may be preferred over incremental validation due to only performing validation once, as opposed to an iterative number of times, which degrades performance. Incremental validating systems perform \(O(N^2)\) validating comparisons, while commit-time validating systems only perform \(O(N)\), where \(N\) is the total number of reads and writes performed by the transaction [24].

Hybrid Systems. The consistency checking models described above do not cover the entire range of possible models. Transactional memory systems can employ hybrid consistency checking models which use various pieces from the above methods. For example, DracoSTM can perform eager acquire write-write conflict detection, while still performing commit-time validation. Likewise, RSTM can perform a combined commit-time invalidation and commit-time validation based on a combination of visible and invisible readers.

3.2 Contention Management: Guerraoui, Herlihy and Pochon

The work of Guerraoui, Herlihy and Pochon has lead to wide adoption of polymorphic contention management. We concentrate on one primary concept in their work: polymorphic contention management systems should allow for user-defined policies. In particular they say, a programmer may also want to implement her own contention managers, for the purpose of her application [13]. While Guerraoui et al. identify the need for user-defined contention management schemes, they focus primarily on run-time alternation of contention managers for optimal performance. Our
work concentrates on extending their prior contributions with a specific focus on building and understanding user-defined contention manager policies.

Polymorphic Contention Management The idea of polymorphic contention management is two-fold. First, a base contention manager class (or interface) exposes a set of interfaces which define how the transactional memory system invokes any given contention manager. Contention managers are then built by implementing some or all of the base class interfaces. Second, the transaction itself contains a pointer to a base contention manager which is invoked internally when the TM systems requires the contention manager to take an action. The base class pointer and common interfaces allows the TM system to invoke specific interfaces for certain conditions which are abstracted away from the concrete TM’s system correctness.

4. DracoSTM’s Extensible Polymorphic Contention Management Interface

Due to the innumerable possibilities of consistency checking model combinations, for the remainder of this work, we narrow our scope of consistency checking to commit-time validation and commit-time invalidation. We do this for a number of reasons. First, validation and invalidation are near opposites and commit-time consistency checking is straightforward in that it is only performed once per transaction. Due to this, we can see the two primary views of consistency checking in an understandable fashion. Second, early TM systems tended toward validation while newer TM systems tend toward invalidation. This makes considering both viewpoints practical. Third, both validation and invalidation perform well under different circumstances. By considering both types, we can help extend their current understanding to user-defined priority-based systems as well. Finally, DracoSTM supports both commit-time validation and commit-time invalidation.

The second type of DracoSTM contention management interfaces is shown in Figure 3. These interfaces implement true management of memory conflicts. The two interfaces shown: abort_before_commit() and permission_to_abort() enable two fundamentally different types of consistency checking. abort_before_commit() is called when DracoSTM is performing validating consistency checking, while permission_to_abort() is called when DracoSTM is performing invalidating consistency checking. Both APIs are called internally at different points in the transaction’s commit phase.

4.1 abort_before_commit()
The abort_before_commit() interface exists to allow a transaction the chance to abort itself before it commits. The abort_before_commit() interface is implemented so a validating transaction can survey all other in-flight transactions before it begins its validation process. If the transaction completes its validation successfully, it will commit. As such, abort_before_commit() is the transaction’s last chance to abort.

The abort_before_commit() interface allows a user-defined priority-based contention manager to walk through all in-flight transactions to determine if an existing transaction has a higher priority than it does. Since abort_before_commit() is called when commit-time validation is occurring, no conflict is identified prior to the call. Furthermore, if a conflict is found after calling the abort_before_commit() interface, the conflict will be unmanageable. Recall that commit-time validation only identifies read-write and write-write conflicts when memory has already been globally committed by a previous transaction. Since these conflicts cannot be undone, the abort_before_commit() offers the CM one final point to abort the transaction before it is committed.

4.2 permission_to_abort()
The permission_to_abort() interface is called when DracoSTM is performing eager invalidation (necessary for direct-updating) or when it is performing commit-time invalidation (necessary for deferred-updating). For this work, we focus on deferred-updating to simplify the complexity of user-defined priority-based transactions.

In the case of commit-time invalidation, permission_to_abort() is called by DracoSTM when a true conflict has been found allowing the contention manager to decide if the attacking transaction is allowed to abort the victim transaction. Both transactions are
5. User-Defined Priority-Based Transactions

Figure 4. Transactions conflicting in commit-time validation.

Let us consider a case where user-defined priority-based transactions are needed. Figures 4 and 5 present two conflicting transactions and how the varying consistency checking models (validation and invalidation) identify the same conflict. The examples show $T_1$ and $T_2$ working on the same integer, $L_0$. In both examples, $T_1$ modifies $L_0$, while $T_2$ reads it. $T_1$ commits before $T_2$ which causes $T_2$ to abort due to an inconsistent read.

Figure 4 shows how the conflicting transactions are handled using commit-time validation. When $T_1$ begins its commit phase, it checks the version of its local $L_0$ data against global memory. It sees the version number is the same and commits its changes to global memory (including incrementing the version). When $T_2$ begins its commit phase, it checks the version of its local $L_0$ data against global memory but sees its $L_0$ version is different than the global version. $T_2$ then aborts and restarts. Figure 5 shows how the conflicting transactions are handled using commit-time invalidation. When $T_1$ begins its commit phase it searches for conflicts with other in-flight transactions. $T_1$ finds a conflict with $T_2$ and flags $T_2$ as invalid. $T_1$ then updates $L_0$ and commits. When $T_2$ performs its next transactional operation (its commit phase), it sees it has been flagged as invalid, aborts itself and restarts.

The commit-time behavior for both Figure 4’s validation and Figure 5’s invalidation is the default CM behavior for DracoSTM, enabling the fastest transaction to commit first. DracoSTM also supports priority-based contention management policies for transactions like Scherer and Scott’s Karma or Polka policies, with priority based on read and write set size [22,23] and Guerraoui et al.’s Greedy policy, with priority based on transaction timestamp [13, 14]. In addition, DracoSTM also exposes a client-controlled transactional priority for user-defined policies.

The transactions shown in Figures 4 and 5 cannot be managed by either DSTM’s Karma or Polka policy, SXM’s Greedy policy or DracoSTM’s default policy such that $T_2$ would commit before $T_1$. Karma and Polka use read and write set size to determine priority and as $T_1$ and $T_2$ have the same size, it is assumed that since $T_1$ reaches the commit first, it would commit first. SXM’s Greedy policy uses timestamp as priority and $T_1$’s timestamp is earlier than $T_2$’s, resulting in $T_1$ committing first. Finally, DracoSTM’s default fastest-first policy allows the fastest transaction to commit first and since $T_1$ is the faster of the two transactions it would be allowed to commit. As such, it is clear that if the user needed $T_2$ to commit before $T_1$, none of the primary CM policies of these three STM systems would suffice. We now turn our attention to building a contention manager that enables $T_2$ to commit before $T_1$.

5.1 Adding User-Defined Priority to Transactions

Consider the requirement of $T_2$ committing before $T_1$ when the two transactions are run simultaneously. This type of system requirement is not unusual. Consider a customer connecting to a web server requesting information about a package she ordered. If we now make $L_0$ the package history and $T_1$ an update on its history and $T_2$ is a status of that history, it is easy to see why a system designer would want $T_2$ to have priority over $T_1$ - reducing customer delays may result in more repeat business. Although this is a seem-
ingly minor requirement, consider the effects if it was not in place and the web server handled business-to-business transactions. The customer checking status could be checking thousands or tens of thousands of orders. If the customer were requesting information about all of her orders, a minor delay in each request could result in an enormous delay for the entire history. As such, the system designer now has a practical reason to make $T_2$ always complete before $T_1$ when run together. Figure 6 demonstrates how this can be achieved in DracoSTM.

```c++

```native_trans<int> global_int;

// Transaction 1's code
void set_global(int val)
{
    for (;;)
    {
        try
        {
            transaction t;
            t.write(global_int).value() = val;
            t.end_transaction();
            break;
        }
        catch (aborted_transaction_exception&)
        {}
    }
}

// Transaction 2's code
void get_global(int val)
{
    for (;;)
    {
        try
        {
            transaction t;
            t.raise_priority(); // <- user priority
            t.write(global_int).value() = val;
            t.end_transaction();
            break;
        }
        catch (aborted_transaction_exception&)
        {}
    }
}
```

Figure 6. Get/set shared integer with user-defined priority.

Figure 6 shows the second transaction ($T_2$) raising its priority so it has a higher priority than the first transaction ($T_1$). While the transactions now have priority, DracoSTM’s contention manager must still be overloaded so it will acknowledge the user-defined priorities. DracoSTM has two CM interfaces for handling conflict, `abort_before_commit()` which is used when DracoSTM is performing commit-time validation and `permission_to_abort()` which is used when DracoSTM is performing commit-time invalidation. Each must be appropriately overloaded depending on the consistency checking model used.

Figure 7 extends DracoSTM’s CM for commit-time, validating, user-defined priorities. The implementation of `abort_before_commit()` iterates through all in-flight transactions, analyzing their priority against the committing transactions priority. If a single in-flight transaction is found with a higher priority, `abort_before_commit()` returns true which causes the committing transaction to abort. Otherwise, `abort_before_commit()` returns false, allowing the committing transaction to proceed.

```c++

```// derived CM abort_before_commit
class val_priority : public base_t
{
    public:
    virtual bool abort_before_commit
    (transaction const &t)
    {
        for (trans_container::const_iterator i =
            in_flight_trans().begin();
            i != in_flight_trans().end(); ++i)
        {
            if (t.priority() < (*i)->priority())
                return true;
        }
        return false;
    }
};

// snippet of validating end_trans code
state transaction::end_transaction()
{
    if (cm_->abort_before_commit(*this))
    {
        lock_and_abort();
        throw aborted_transaction_exception
        ("aborting due to CM priority");
    }
    // ... rest of end transaction here
}
```

Figure 7. A Validating, Priority-Based, Transaction Scheduler.

Figure 8 extends DracoSTM’s CM for commit-time, invalidating, user-defined priorities. In Figure 8, the committing, attacking transaction first identifies a true memory conflict. DracoSTM’s conflict handler then calls into the CM’s `permission_to_abort()`, passing both the attacking and victim transactions as parameters to the method. The user-defined `permission_to_abort()` then returns true if the attacking transaction has an equal or greater user-defined priority. In the case of the prior get and set transactions, the get transaction ($T_2$) has a higher priority and would therefore force the set transaction ($T_1$) to abort itself due to a negative response from `permission_to_abort()`.

5.2 Summary

This section demonstrated how DracoSTM’s commit-time validation and invalidation contention management system could be overloaded to handle simple user-defined priority-based transactions. An important observation regarding commit-time validation is that it does not detect actual conflicts. As such, DracoSTM’s `abort_before_commit()` implementation as shown in Figure 7 does not abort a transaction based on a true memory conflict. Instead, commit-time validation aborts all committing transactions that have a lower priority than any in-flight transaction, even if no true conflict exists. Commit-time invalidation detects only true conflicts. Therefore, DracoSTM’s `permission_to_abort()` implementation as shown in Figure 8 only aborts transactions when true memory conflicts exist. For the example described in this section, both consistency checking schedulers work equally as well.
class val_priority : public base_t
{
  public:
      virtual bool permission_to_abort
      (transaction const &lhs,
        transaction const & rhs)
      {
        return lhs.priority() >= rhs.priority();
      }
    
    // invalidating write-write conflict code
    void transaction::abort_write_write_conflicts()
    {
      // iterate through all our written memory
      for (iter i = w().begin(); w().end() != i; ++i)
      {
        // iterate through in flight transactions
        for (iter j = in_flight_trans().begin();
             j != in_flight_trans().end(); ++j)
        {
          // memory conflict?
          if (j->w().end() != j->w().find(i))
          {
            if (cm_->permission_to_abort(*this, *j))
              // add to flag_as_aborted list, flag
              // only after all conflicts are found
              throw aborted_transaction_exception
                ("abort due to CM priority");
            else
              // iterate through all our written memory
              for (iter i = w().begin(); w().end() != i; ++i)
              {
                // iterate through in flight transactions
                for (iter j = in_flight_trans().begin();
                     j != in_flight_trans().end(); ++j)
                {
                  // memory conflict?
                  if (j->w().end() != j->w().find(i))
                  {
                    if (cm_->permission_to_abort(*this, *j))
                      // add to flag_as_aborted list, flag
                      // only after all conflicts are found
                      throw aborted_transaction_exception
                        ("abort due to CM priority");
                  } // else
              } // else
          } // if
        } // for
      } // for
    } // abort_write_write_conflicts

Figure 8. An Invalidating, Priority-Based, Transaction Scheduler.

6. Growing Complexity of User-Defined Priority-Based Transactions

In the below example, we present the need for a TM system that guarantees more than starvation prevention. Although the below example is applied to a specific domain, the same class of problem exists in a number of other areas, such as: medical, space and defense systems. In each of these area response time must be guaranteed within rigid parameters or drastic consequences may result (e.g. irreversible patient damage or death, spacecraft failure and citizen or military loss of life). As such, the below demonstration of strict behavioral requirements can be considered as applicable to a variety of critical systems.

The example problem is implemented by three interactive threads:

1. Thread 1 (transaction T1) iteratively reads each element of the integer array in a single transaction.
2. Thread 2 (transaction T2) executes one of several high-priority transactions based on the values read from thread 1. In addition, at a very infrequent rate, T2 sometimes rereads the shared integer array.
3. Thread 3 (transaction T3) updates one integer element in the shared integer array as updates occur.

The above scenario could be applied to a number of different practical examples. One such scenario is to consider the three threads working together as an automated stock market exchanger based on real-time trends. Each integer location represents a stock market value. Thread 1 and 3 perform relatively straight forward actions. Thread 1 makes a local copy of all the current stock values and passes them to thread 2. Thread 3 updates a specific stock in the shared array with the latest real-time data.

Thread 2’s behavior, however, is more complex. Thread 2 uses the results sent to it by thread 1 to perform two different actions; (1) sell a stock or (2) buy a stock based on the comparison of the current values against historical data. Thread 2 always queries and processes the historical data before taking any action. The querying and comparison of historical data takes a certain amount of time. Generally the actions performed by thread 2 are of relatively minimal consequence, so the delay in processing the historical data has only a minor negative affect on the command being completed. However, in some special cases, when the action being taken by thread 2 is of a critical nature (e.g. selling or buying large amounts of stocks) a reverification of the current stock values is needed in order to ensure the delay in processing the historical data has not resulted in an unacceptable change in stock values. In these cases, thread 2 must reread all of the shared integer array values and validate them again before proceeding.

T2’s natural priority is very high. If T2’s execution is not performed quickly, system actions become less correct. Furthermore, if the shared integer array size is large, T2’s array read operation may be starved by T3’s write operation. T1 would be starved by T3 due to T1’s read operation performing substantially more work than T3. T3’s single write on the shared integer array is guaranteed to conflict with T1’s read, ensuring a conflict exists between the two transactions each time T3 commits. If T1 and T3 are run continuously, the prior static priorities of Figure 6 would no longer suffice. The prior static priorities are no longer sufficient because T2 will continually commit before T1, resulting in either T1 aborting if T3 has a higher static priority or T2 aborting if T2 has a higher static priority. As such, a different priority-based solution must be implemented. While either Scherer and Scott’s Karma policy or Guerraoi et. al.’s Greedy policy would prevent starvation between T1 and T3, neither would allow T2 to take a natural higher priority than either T1 and T3. Since some cases exist where T2 conflicts with T3, some mechanism must be put in place to prevent T2 from committing when it is conflicting with T2. Thus, we need user-defined priorities for this problem to ensure system correctness.

6.1 Dynamic Priority Assignment

A scheduling model where each task is prioritized based on how close it is to its deadline is called dynamic priority scheduling or assignment [25]. Dynamic priority assignment is preferred over the static priority assignment shown in Figure 6. We use a basic form of dynamic priority assignment for transactions T1, T2 and T3 which increase their priority each time they are aborted. Our dynamic priority scheduling is similar to Ramanathan and Moncef’s dynamic priority based scheduling (although considerably simplified) [31]. In Ramanathan and Moncef’s dynamic priority algorithm, they increase priority as deadlines grow closer, we increase priority based on the number iterative transactional priority-based aborts. Both algorithms have the same fundamental underpinnings: as time moves forward and the task fails to complete, the priority of the task increases. Ramanathan and Moncef’s algorithm functions this way in order to meet time-critical deadlines, our algorithm does this to prevent starvation between transactions T1 and T3.

The DracoSTM code used for all three transactions is shown in Figure 9. Transaction T1 copies the shared integer array. Transaction T2 simulates a random prioritized task (10-99 priority) and ar-
native_trans<int> global_int;
native_trans<int> arr[100];

// T1 copies all data from arr
void get_arr(int out[]) {
    for (transaction t ;; t.raise_priority())
    {
        try {
            for (int i = 0; i < 100; ++i)
            {
                out[i] = t.read(arr[i]).value();
            }
            t.end_transaction();
            return;
        }
        catch (aborted_transaction_exception&)
        { t.restart_transaction(); }
    }
}

// T2 executes a task of priority 10-99, writes
// and reads to global_int and rand() sleep time
int exe_task(int v, int orig[]) {
    transaction t;
    t.set_priority(10 + rand() % 90);
    for (;; t.raise_priority())
    {
        try {
            int ret = t.read(global_int).value();
            t.write(global_int).value() = v;
            sleep(rand() % t.priority());
            // rarely revalidate array
            if (0 == rand() % 1000) {
                int out[100];
                get_arr(out);
                // verify out and orig are the same
                t.end_transaction();
                return ret;
            }
        }
        catch (...) { t.restart_transaction(); }
    }
}

// T3 executes set_arr iteratively
void set_arr(int val, int loc) {
    for (transaction t ;; t.raise_priority())
    {
        try {
            t.write(arr[loc]).value() = val;
            t.end_transaction();
            break;
        }
        catch (aborted_transaction_exception&)
        { t.restart_transaction(); }
    }
}

Figure 9. Transactions $T_1$, $T_2$ and $T_3$.

6.2 Priority-Based Transactions & False Positives

$T_3$, as shown in Figure 9, begins with a high initial priority. $T_2$ performs most of its reads and writes in isolation. Therefore, $T_2$ very rarely creates true transactional memory conflicts. However, if commit-time validation is used when $T_2$ is run the system will abort all committing transactions which have a lower priority than $T_2$, although most transactions will be free of memory conflict. Although some true conflicts exist, most of the required system aborts of $T_1$ and $T_3$ while $T_2$ is executing do not truly conflict with $T_2$. The cost of these additional false positive aborts is a heavy performance penalty shown in the experimental results section.

The invalidating system’s priority scheduler shown in Figure 8 processes $T_2$ with no false conflict identification. DracoSTM’s commit-time invalidating contention manager scheduler is only invoked when a true memory conflict is found. As such, even in the cases when $T_2$ must read the shared integer array but has yet to perform the reads, updates to the shared integer array by $T_3$ do not cause a conflict. When $T_2$ has already read the shared integer array and $T_3$ tries to update a location and commit, $T_3$ will be forced to abort as to prevent it from inverting $T_2$’s priority. From this context, the commit-time invalidation system runs in more efficiently and remains fully correct. The commit-time invalidating scheduler will not invert any transaction’s priority and, simultaneously, ensures no transactions are incorrectly aborted due to falsely identified conflicts. The results of this more correct behavior yield significant overall performance benefits as shown in the experimental results section.

6.3 A Critical Observation

A critical observation derived from the above scenario is that any system that does not perform full invalidation (either eagerly or lazily) must abort all lower priority, committing transactions when higher priority, in-flight transactions exist. Lower priority transactions must be aborted in non-full invalidating systems, because the system is incapable of identifying true memory conflicts. Since true conflicts cannot be identified prior to a transaction committing, lower priority transactions which may adversely affect higher priority, in-flight transactions must abort themselves in order to ensure priority inversion does not occur.

Consider a system which performs eager invalidation with invisible readers. Eager invalidation with invisible readers requires the readers validate themselves at commit-time. When invisible readers reach their commit phase, cases can exist when they must abort due to a previously committed write conflict. If such a case occurred, such that a high priority, invisible reader was aborted due to a previously committed lower priority writer, the system would exhibit priority inversion. To prevent this, any transaction that is not performing complete invalidation, must survey all in-flight transactions before committing due to the potential that a conflict exists with its transactional state and another in-flight transaction with higher priority. If a higher priority, in-flight transaction is found while the committing transaction is iterating through the in-flight transaction list, the committing transaction must abort.

The affects of this observation restrict general contention management behavior for systems that require user-defined task ordering. Prior contention management systems have been unrestricted
in that they have allowed portions of the consistency checking system to be performed by both validation and invalidation. The mixed validation and invalidation portions of fairness-oriented contention managers has been reasonable as the invalidated portions of the transaction help to identify manageable conflicts, while the validated portions of the transaction help scalability. Thus, contention managers that are driven toward fairness have previously allowed a mixture of validation and invalidation and have been reasonable to do so. Yet, user-defined priority-based transactions require more strict partial ordering commit behavior that fairness driven contention managers cannot achieve. A primary reason for this shortcoming is that validation suffers universal performance deficiencies for user-defined priority-based transactions due to the strict ordering requirement. These performance penalties are the direct result of false conflicts leading to a high number of aborted transactions. As such, it is possible that a re-evaluation of contention management philosophy for systems that require user-defined priority-based transactions may be necessary. However, a detailed analysis of this problem is outside the scope of this work.

7. Experimental Results

The experimental results gathered were based on running concrete instances of the previously explained three threaded example in the prior section shown in Figure 9. All results were run on a 3.2 GHz 4-processor Intel Xeon with 16 GB of RAM. A brief summary of each thread’s workload is listed below.

1. Thread 1 (transaction $T_1$) reads each element of the integer array in a single transaction.
2. Thread 2 (transaction $T_2$) simulates the execution of a high priority task by creating a high priority transaction, sleeping for a time and writing to memory different than $T_1$ and $T_3$.

In some cases, $T_2$ must reread the shared integer array.

3. Thread 3 (transaction $T_3$) updates one integer element in the shared integer array as updates occur.

Program termination was controlled by the total number of successful iterations of $T_3$. When $T_3$ had successfully completed $N$ times, the program terminated. Any number of program termination mechanisms could have been used, such as, total duration, successful iterations of any thread’s transaction, total number of aborts, etc. We chose $T_1$’s successful completion arbitrarily.

Three sample iterations of $T_1$’s required workload were run, $10^3$, $10^2$ and $10^1$ for both commit-time validation (labeled: $\text{val}_{10}$, $\text{val}_{100}$ and $\text{val}_{1000}$ in the graphs) and invalidation (labeled: $\text{inval}_{10}$, $\text{inval}_{100}$ and $\text{inval}_{1000}$ in the graphs). In addition, for each set of iterations, $10^3$, $10^2$ and $10^1$, three different shared array sizes were used: $10^3$, $10^2$ and $10^1$. The result is a nine-way performance analysis (three shared integer array sizes by three different $T_1$ successful completions) per consistency checking model. The varying array sizes were used in conjunction with the varying $T_1$ iterations to ensure results of a specific consistency checking model benchmark were not a side-effect of a specific shared array size, particularly when considering array-cache alignment.

The results shown in Figures 10(a) and 10(b) are as expected based on the prior theoretical analysis of commit-time validation and invalidation. Commit-time validation performs roughly one to two orders of magnitude worse than commit-time invalidation due to its inability to abort only true memory conflicts. Furthermore, commit-time validation suffers roughly a 25-40 abort-to-commit ratio with two outliers of roughly 8 and 15 abort-to-commit ratios as shown in Figure 10(b). Commit-time invalidation consistently has less than 0.008 abort-to-commit ratio for all benchmarks, making them invisible on the graphs in Figure 10(b). Abort-to-commit ratio is the key performance controlling statistic of this example. Commit-time validation performs poorly because of its high abort-to-commit ratio, while commit-time invalidation performs well because of its low abort-to-commit ratio.

8. Conclusion and Future Work

We extended the prior work of Scherer and Scott and Guerraoui et al. by implementing user-defined contention managers. We found consistency checking models have a direct effect on the correctness of building user-defined priority-based transactions. Ensuring strict partial ordering of competing tasks is a stronger requirement than is achieved by fairness-oriented contention managers and, as such, some of the previous contention management flexibility is lost. The more strict requirements of user-defined priority-based transactions cause splintering performances differences between consistency checking models. The significant performance differences of scheduling user-defined priority-based transactions in commit-time validation and invalidation were presented in experimental results section. Invalidation performed one to two orders of magnitude more work than validation.

We identified a critical observation such that any system which supports user-defined priority-based transactions and does not use either full eager or lazy invalidation must abort any committing lower priority transactions if other higher priority transactions are in-flight regardless of whether these in-flight transactions present true memory conflict. The degraded performance caused by falsely aborted transactions suggests a refocus of contention management philosophy for critical systems and perhaps the prohibition of validation consistency checking in such systems.

Commit-time invalidation is the only consistency checking model we found to identify true conflicts, yet it comes with a
heavy performance penalty when in-flight, competing transactions are high. As such, the remaining speculative invalidation models should also be considered as alternative consistency checking possibilities for critical systems. Also, the possibility of run-time alternation of consistency checking is another open question that should be analyzed further. While we have attempted to present the foundations of why such behavior is required, the user-defined priority-based contention management policies we presented were extremely basic. Clearly, a number of open questions remain regarding the analysis of real-time, critical or deadline-driven system behavior in conjunction with user-defined priority-based transactions. An abundant body of research already exists in task scheduling and, as such, alternative solutions should be explored and compared against our findings.

9. Acknowledgements

We would like to acknowledge Mark Holmes, Randy Richmond, Paul Rogers and Don Searcy of Raytheon Company for their support of our transactional memory research and for improving our understanding of critical systems. We would also like to thank the anonymous reviewers for their feedback. The anonymous reviewers feedback has been critical in helping us focus our ideas and significantly improve the quality of this work.

References


