The Xfork in the Road to Coordinated Sibling Transactions

Hany E. Ramadan and Emmett Witchel
Department of Computer Sciences, University of Texas at Austin
{ramadan,witchel}@cs.utexas.edu

The architectural shift towards multi-core processors is focusing renewed attention on parallel programming. Transactional memory (TM) is a promising paradigm for making parallel programming easier. Programmers simply define the parallel work, and atomic execution is handled by the TM system. In this paper, we pose the next question: Can programmers find parallel work *within* their transactions? And, if so, can they exploit this parallelism easily? This paper answers in the affirmative on both counts.

This paper presents the notion of *coordinated sibling transactions*, a more powerful and expressive generalization of the traditional model of parallel closed-nested transactions. Coordinated siblings transactions are accessed via a new API we define, the *xfork* primitive, which is designed to make parallel closed-nested transaction a general commodity. This API allows programmers to specify the *forms* of coordination among the siblings, while not worrying about how these semantics are implemented by the runtime. Programmers can thus structure concurrent work in ways which would be extremely burdensome, if not impossible, to achieve on existing systems.

This paper also presents the design and implementation of sibling STM (SSTM), the first STM to implement these features. SSTM is built on the .NET Common Language Runtime (CLR) [17], so we discuss how this runtime's features (transaction and threading libraries) can be leveraged to produce a parallel nested STM system. The evaluation of our prototype demonstrates that programmers, using xfork to express intra-transaction parallelism, can improve transaction performance.

1. Introduction

> Intratransaction parallelism requires genuine support for nested transactions.
>
> -Jim Gray and Andreas Reuter, *Transaction Processing*

The architectural shift towards multi-core processors is putting renewed attention on parallel programming. Transactional memory [14] (TM) has attracted attention for its promise of a parallel programming model that is easier to use than locks, with performance that is just as good. To date, TM has addressed only synchronization of threads, not how a program can use more threads to increase throughput or decrease latency.

The traditional prescription for intra-transaction parallelism is a nested subtransaction model, extended to allow the nested transactions to run in parallel. This basic model works only if the parallel activity within the transaction is independent. However, the fact that the work was put into the same transaction to start with makes complete independence unlikely. Moreover, some kinds of parallelism, such as speculation, are not exploitable in this model. While this intra-transaction model is conceptually simple, it has not been widely used in a database context, or examined in much depth in the newer TM context.

We suggest that *non-independent* parallel closed-nested transactions are easy to program and can improve performance. Recent TM systems have looked at linear (non-parallel) closed-nesting [18, 19], but found its performance lacking [18]. Parallel nesting is as old as nesting itself, and has also been proposed recently in the TM context [1]. However, Agrawal's model treats each nested transaction as completely independent of its siblings. We believe this model insufficiently expresses some of the intuitive sources of parallelism, and should be complemented with siblings that can affect each other's outcomes.

This paper introduces **xfork**, a programming construct designed to make it easier for programmers to express intra-transaction concurrency. It enables programmers to leverage additional cores to increase performance, while retaining the ease of use of the TM API. Xfork is the mechanism by which programmers create **coordinated sibling transactions**. These transactions are similar to parallel closed-nested transactions, but with added coordination semantics that make them more accessible and useful to programmers.

Coordinated sibling transactions allow programmers to treat parallel nested transactions as a group and to specify the semantics for the group as a whole. Real life shows that some siblings are independent, some act as one unit, and some cannot survive together. We formalize this relationship in our model as the *sibling coordination forms* OR, AND, and XOR, using a rough analogy to the Boolean functions. We show the utility of these forms for TM programmers and the natural fit with TM code structure.

The xfork API can be viewed as a blend of nested transactions, fork/join parallelism, distributed transaction coordination and speculation. If programmers are to focus on their application's algorithms, and parallelism within them, the runtime should shield them from the complexities of the low-level implementation of such coordination. The xfork implementation handles the threading and concurrent execution of the different forks, and performs the necessary coordination. Programs built with the xfork API benefit because they can use extra cores to speed up individual transactional units of work.

The contributions of this paper are as follows:

1. We introduce the xfork API and the coordinated sibling transaction model (Section 3), after a motivating example (Section 2) that demonstrates the multiple ways they can be used.
2. We explore issues related to building a TM system on the Common Language Runtime, and present the architecture of SSTM, one such system (Section 4).
3. We present the detailed design of our implementation of the coordinated sibling model (Section 5) and an STM implementation that supports parallel closed-nesting (Section 6).
4. We evaluate the ability of our prototype to scale the performance of several benchmarks (Section 7).

The remainder of the paper discusses related work (Section 8) and then concludes (Section 9).

2. Example

This section illustrates coordinated sibling transactions with an example taken from one of the benchmarks in our evaluation. As a nod to the recent global attention paid to financial sectors, our example is about speeding up financial transactions.

2.1 Background

Our example is the prototypical bank transfer scenario, where a sum of money is transferred between two bank accounts. The bank accounts are represented by linked lists that need to be protected from concurrent accesses.

We'll assume that the two bank accounts are stored in separate lists (one for checking accounts and one for savings accounts). If each list is protected by a lock, and the transfer operation does not use fixed lock ordering, then concurrent transfer operations can result in deadlock. Deadlock occurs when each of two transfers acquires one data structure's lock and waits for the lock held by the other operation.

Transactions simplify the programmer's task because he or she need not consider the order in which data structures are accessed. Thus, the potential for programmer-visible deadlock is eliminated, as is the complexity involved in avoiding deadlocks. In the example of two data structures and two locks, lock ordering is trivial. However, when scaled to real programs, lock ordering becomes difficult, sometimes even impossible, to define and maintain.

2.2 Data structures

Our basic example starts with two data structures. The first, checking-list, holds checking account records, with one record per node. The second, savings-list, holds savings account records. To make the example even more flexible (and more realistic), we add a third structure that is also accessed by the transfer operation. We assume that if the amount transferred into a savings account exceeds a certain threshold, then the bank wants to make a note of the transaction. For our example, the bank may want to send the customer involved a special offer to upgrade his savings account to silver or gold status. Since the accounts involved in this list are called Notable Savers Accounts, we call this third structure the nsa-list. All three structures are doubly-linked lists.

2.3 Transactional transfer code

The pseudo-code for the basic transactional transfer operation as it might be implemented on STMs today is shown in Figure 1. We make a few observations:

- The account numbers are specified in the src-accno and dst-accno parameters. For clarity, we assume both accounts always exist, and omit the code to deal with missing accounts.
- The destination account will be credited if and only if the source account debit succeeds (i.e. there are sufficient funds).
- We want to add the accounts to the nsa-list even if funds were insufficient to complete the transfer because, for the bank's purposes, attempted transfers are just as important to track as actual transfers.

We omit the pseudo-code for SearchList, which is a simple linked list traversal. The AddNotableList code searches the list to see if the record for the specified account exists and, if so, notes the latest transfer amount in that record. If the account isn't already notable, it adds a new record for the account to the list. The entire transfer operation is performed within a single transaction. The code executes safely even in the presence of other concurrent transfer operations. Without transactions, the code would be more complex, due to locking issues and recovering from partial failures. The code is simple and correct, but for programmers wanting to take advantage of multi-core processing, the basic TM API provides no further easy routes for improving the latency of the transaction. The next subsection shows how sibling transactions can improve this code's performance with very little coding effort.

```
Transfer (src-list, src-accno, amount
          dst-list, dst-accno, nsa-list)
{
  atomic
  {
     DebitCredit(<params>)
     NsaCheck(<params>)
  }
}

DebitCredit (<params>)
{
  src-account = SearchList(src-list, src-accno)
  if (src-account.balance >= amount)
  {
    src-account.balance -= amount
    dst-account = SearchList(dst-list, dst-accno)
    dst-account.balance += amount
  }
}

NsaCheck (dst-list, dst-accno, nsa-list, amount)
{
  if (dst-list == savings-list && amount > 10000)
    AddNotableList (nsa-list, dst-acct, amount)
}
```

Figure 1. Traditional transactional code for the transfer operation

2.4 How to parallelize the transfer operation

Even though our transfer operation is trivial in size, concurrency can be improved with the appropriate tools. We can identify three potential areas for concurrency, each of which corresponds to one of the forms of sibling transactions. We will explain the forms in detail in Section 3, but intuitively these determine under which conditions sibling transactions are allowed to commit.

- The actual debit/credit is independent of the nsa-list. Regardless of whether the debit/credit succeeds, we will be performing the nsa-list check. Thus, the nsa-list work can be done concurrently. This is an example of OR-form sibling transactions. (The Boolean notation describes the relation between the success of the forks, and the success of the xfork operation). Note that there is no short-circuit evaluation in OR-form, each of the forks are executed.
- The actual transfer consists of two parts, debit and credit. They can be performed concurrently, since the credit part does not depend on data from the debit. The only restriction is that the credit part must not be allowed to complete if the debit fails (i.e. insufficient funds). This is an example of AND-form sibling transactions. There is short-circuit evaluation in this form.
- The SearchList function, which operates directly with the linked lists, may also be sped up by performing the search in multiple ways. Because we are dealing with doubly-linked lists, the search may be performed simultaneously from both ends of the list (i.e. both forward and backward traversal). This kind of parallelism is data structure and operation specific. This

```
Transfer (<params>)
{
 atomic
 {
   xfork (OR, 2, {DebitCredit, NsaCheck},
                 {<params>, <params>}
         )
 }
}

DebitCredit (<params>)
{
   xfork (AND, 2, {AcctAccess, AcctAccess},
            {(src-list, src-accno, -1*amount),
             (dst-list, dst-accno,amount)}
         )
}

AcctAccess (list, accno, amount)
{
    account = SearchList(list, accno)
    account.balance += amount
    if (account.balance < 0)
       return Failure;
    else
       return Success
}

SearchList (list, accno)
{
    var account
    xfork (XOR, 2, {FwdSearch, BackSearch},
              {(list, accno, &account),
               (list, accno, &account)}
          )
   return account
}
```

Figure 2. Parallel transactional code for the transfer operation, using sibling transactions

parallelism is an example of XOR-form sibling transactions. (Even if both succeed only one should commit).

2.5 Parallel transactional transfer code

Taking into account the potential concurrency identified in the previous section, we can re-implement the transfer operation to take advantage of sibling transactions. The new code is shown in Figure 2.

The pseudo-code uses xfork with four parameters: the first specifies the form of coordination between the sibling transactions (AND, OR XOR), the second specifies how many sibling transactions are to be created, the third specifies the procedures to be invoked within the nested transactions, and the fourth defines the parameters to pass to each procedure. The next section describes the API in more detail.

The changes are straightforward and localized. The transfer operation itself now simply uses an OR-form xfork to execute both the transfer operation and the NsaCheck concurrently. The SearchList procedure is rewritten to use an XOR-form xfork to search a list concurrently using forward and backward traversal (we omit the FwdSearch and BackSearch traversal code). Finally, the DebitCredit procedure now uses an AND-form xfork operation to perform both parts of the operation concurrently. A new procedure, AcctAccess, is invoked to perform both the debit and credit operations. It returns a failure code (which aborts the operation) if the balance goes below zero (i.e. insufficient funds). The AcctAccess procedure invokes SearchList, which also uses xfork. The AddNotableList function (called by NsaCheck), is unchanged, though it benefits from the modifications to SearchList.

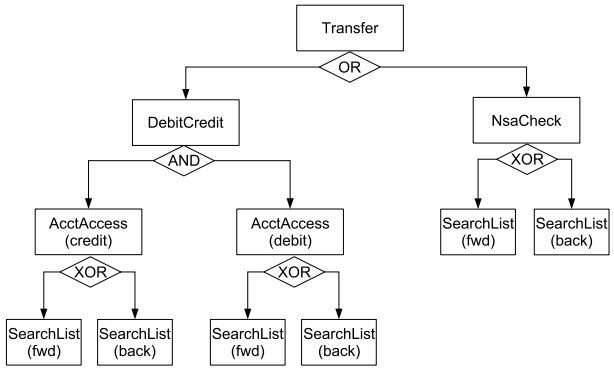

Figure 3. Sibling transactions created by transfer example. The diamonds represent xfork invocations, of the specified types.

2.6 Observations about the modified code

Our modified example code highlights several important aspects of sibling transactions:

- Sibling transactions are composable. A transaction can create siblings with xfork and these sibling transactions can create nested siblings themselves. Figure 3 illustrates the sibling transaction tree produced by the execution of the transfer operation. The six leaf nodes represent the maximum number of concurrent threads that can be involved in executing the single Transfer operation.
- Sibling transactions retain the safety property of transactions. When identifying potential areas for concurrency, we focus on the parts of the original algorithm that are independent, and thus can safely execute concurrently. However, since the transactions execute as nested subtransactions the programmer does not need to guarantee that they are completely independent of each other. If they touch common data, the fact that they are sub-transactions provides the atomicity property that protects them from concurrent accesses by other siblings (or, for that matter, from other top-level transactions).
- Of the three forms of xfork in this example, only the XOR form involves speculation. This form of speculation is novel because it uses the atomicity and rollback properties of transactions to allow the programmer to express speculative execution paths that cannot be automatically inferred.

3. Coordinated Sibling Transactions: Model

This section presents the xfork API and the semantics of coordinated sibling transactions.

3.1 Public API

We add a single function, *xfork*, to the TM API which provides coordinated sibling transactions. The funcion name stands for transactional fork, and builds on intuitive notions of the fork/join pattern for expressing parallelism (and to a lesser extent the fork system call). The method signature from our C# implementation is shown

```
bool xfork (xforkForm form,
                  int numForks,
                  xforkProcedure forkProc,
                  object data);

enum xforkForm { AND, OR, XOR };

delegate xforkResult xforkProcedure (int forkNum,
                            object data);

enum xforkResult { Success, Failure };
```

Figure 4. The xfork API and accompanying types. The user implements the actual sibling code in a function with the signature of xforkProcedure.

in Figure 4. Other languages, or overloads, can provide additional syntactic sugar for the API (e.g. separate forkProcs for each fork), but the core parameters for xfork are as follows:

- *form*: the form of sibling coordination (AND, OR, XOR).
- *numForks*: the number of concurrent sibling transactions to create.
- *forkProc*: a procedure to execute inside the sibling transactions.
- *data*: (optional) user-specified data to be passed to each forkProc.
- *return-code*: the xfork function returns true on success, false otherwise.

The system will create numForks nested transactions, with implementations free to schedule them as concurrently as possible. The system starts the sibling transactions before it calls into the user-provided forkProc, passing in the optional data. This data usually contains work-partitioning information, space for output results, and so on. Fork procedures (forkProcs) are passed the forkNum parameter, to identify which sibling fork is being executed on this thread.

The forkProc returns a code indicating success or failure, which is used to determine the fate of that transaction (and possibly the fate of the entire set of siblings). Note that forkProcs do not themselves commit or abort the sibling transactions they are executing in—the commit decision is handled by the system. Sibling transactions may request a restart, as any regular transaction can, and they may also restart due to conflicts with other transactions. Re-execution after restarts will cause the forkProc to be invoked again by the system.

3.2 Sibling forms

The form parameter specifies the coordination semantics between sibling transactions. We focus on the three core forms used by our benchmarks (though our system allows custom user-defined forms). The semantics of each form is a function of the return codes of the forkProcs:

- AND: All sibling transactions must succeed, or none succeed.
- XOR: Only one sibling transaction must succeed.
- OR: Sibling transactions succeed or fail independently.

The coordination happens in a (conceptual) pre-commit phase for the sibling group. The xfork operation decides the outcome for each sibling transaction based on the form semantics. If the semantics are satisfied, then the siblings commit as allowed by their form type (e.g. in the AND form all siblings commit, in the XOR form only one could commit, and in the OR form all successful siblings commit).

All sibling transactions need not complete execution before a decision to commit is made. An XOR-form call completes as soon as a single sibling return success and finishes commit. The system need not wait for other siblings to complete, and ensures that they will abort (if it has scheduled them). In an AND-form call, if a single sibling returns a failure code, the parent fails.

These forms emulate a wide range of coordination behaviors, ranging from the traditional distributed-transactions semantics of the AND form, to the traditional independent closed-nested transactions of the OR form, to the speculative nature of the XOR form. However, the XOR form can also be used in a non-speculative manner, such as when a work item is known to be in one of a set of data structures (or buckets). In that case, parts of data structures can be searched in parallel, but as soon as one transaction finds the target, the XOR semantics terminate the other siblings.

3.3 xfork return code

The xfork API returns true if siblings succeed according to the specified form's semantics, and false if they do not. For example, the XOR form will return false if none of the forks return success, while the AND form will return false if any fork returns failure. When xfork returns false, it guarantees to the caller that *no siblings will commit* (i.e. system state is the same as before the xfork call).

If the xfork operation completes successfully, it returns true. The read and write sets of the successful siblings are merged into the parent transaction following the usual rules of closed-nesting. For the OR form, an overload of the API can return status flag which indicates which siblings (forks) completed successfully, and which failed.

Edge cases exist where the semantics associated with a true or false return code cannot be provided. The implementation detects these cases before returning a result to the user, and handles them by forcing the parent transaction to restart. This case arises due to the inability to *guarantee* an atomic commit of a set of siblings, even after xfork successfully completed the prepare phase of the entire set of transactions. The final decision rests with the CLR's transaction manager (discussed below), and possibly other resource managers. By relying on higher-level failure-handling (aborting the parent transaction), we guarantee that the transaction is always executing in a well-defined program state. This is one of the key reasons why xfork can only be called from within a transaction.

3.4 Sibling conflicts

Programmers using the AND form should ensure that the sibling transactions have independent write sets (xfork returns an error and aborts all siblings if this is violated). They may conflict with non-sibling transactions, but it is a programmer error if they conflict with each other. It may be possible to run such conflicting siblings serially, but until we have more experience with AND-form siblings, it is unclear whether this situation is one that should be supported. Conflicts among siblings in the OR or XOR form are allowed, and will cause one of the siblings to restart. If the conflict occurs with a sibling that has executed successfully (but hasn't committed, since the entire xfork is still active by definition), then such a conflict will cause the non-completed sibling to behave as if it has returned a failure error code. The thread stops attempting to re-execute. In related work, we discuss how techniques, such as dependence-aware transactions, can be used between siblings to eliminate certain classes of conflicts.

4. SSTM: Architecture

This section describes the architecture of sibling STM (SSTM). SSTM is our prototype implementation of an STM that supports

the xfork API and coordinated sibling transactions. It is built on top of the .NET Common Language Runtime. This section provides an overview of how we use and extend the CLR and integrate with its transactional libraries. This section also provides an overview of the two primary components of SSTM: the coordinated sibling executive (SibEx) and the nesting-aware transactional object store (TxStore). The two components are mostly independent[1] and the detailed design of each will be presented in Section 5 (SibEx) and Section 6 (TxStore).

4.1 Common Language Runtime

The Common Language Runtime (CLR) is Microsoft's managed execution environment that underlies several commercial and research languages. The CLR has several features that make it an attractive environment for our prototype work. These include significant transactional and threading infrastructures, located in the System.Transactions and System.Threading class libraries. Our prototype can be used by any CLR language (as long as we stay within the bounds of the CLI specification), including C# and VB.Net.

Our design is somewhat constrained because we do not have access to the CLR source code and we did not want to make changes to the C# language. Thus, we built on top of the CLR, exploiting as many extensibility points as possible while working within the C# language. The ability to change the language or the internals of the execution engine would result in even tighter integration of sibling transactions.

4.2 CLR transaction model terminology

SSTM works within the database-derived model defined by the System.Transactions library, a model different from that used by most STM implementations. Therefore, some of the terminology may be unfamiliar in the TM context. Usually an STM system provides a programming interface (such as the atomic keyword) to use transactions, and a runtime implementation that manages the transactions and memory.

Our database-derived model separates several of the system's components. The *transaction manager* (TM) is responsible for tracking active transactions, qua transactions, and coordinating their commit/abort. The TM is not directly involved in the execution of operations inside any given transaction. One or more *resource managers* (RM) manage the actual resources that may be transactionally modified, as well as export the operations that make these modifications. The TM and RM must be coordinated. This coordination is handled through the two-phase commit (2PC) protocol, where RMs vote on whether a transaction should commit, and are informed of the outcome of the transaction as determined by the TM. The first time that a transaction uses an RM, the RM must *enlist* in the transaction, meaning that the RM contacts the TM to participate in the 2PC protocol.

SSTM uses this model, with the actual transactional memory being represented by the TxStore RM. Moreover, the coordination between sibling transactions (performed by the SibEx component) does not require modification of the TM, but can be implemented as just another RM involved in the transaction. Unlike existing STMs, which usually provide their own set of TM APIs and do not support additional resource managers, SSTM must cope with the loss of absolute control over, for example, the process involved in starting and completing transactions.

4.3 System.Transactions interfaces

In the CLR, the TM is provided by the System.Transactions library, while the user must implement the RMs. SSTM's two main components (TxStore and SibEx) are resource managers, and thus must interact with the main RM interface, an interface responsible for enlistments. An enlistment is made using the current transaction object's EnlistVolatile method (Transaction.Current is a thread-local variable that holds the current transaction object, if any). An enlisting RM must implement an interface that is used as part of transaction completion. The IEnlistmentNotification interface (defined in System.Transactions) is shown in Figure 5. The interface reflects the familiar two-phase protocol commonly used by distributed transaction managers[2].

```
interface IEnlistmentNotification
{
  void Commit (Enlistment enlistment);
  void InDoubt (Enlistment enlistment);
  void Prepare (PreparingEnlistment
                  preparingEnlistment);
  void Rollback(Enlistment enlistment);
}
```

Figure 5. The System.Transactions.IEnlistmentNotification interface, implemented by both TxStore and SibEx.

System.Transactions also includes the interface that is used by the programmer to create transactions. Transactions are created through the TransactionScope mechanism. This mechanism is similar to the familiar atomic block with a few exceptions, notably that transactions do not automatically retry when they abort. Transaction commit or abort is ultimately determined by the TM and all the RMs together, not by any individual RM. This framework has the benefit of being more general and extensibile as compared to existing STM designs.

Finally, the System.Transactions framework does not directly support nested transactions. We therefore define our own convention for managing closed-nested transactions. We define a property, *parents* using the CLR's thread-local storage facility. The parents property holds a list of parent Transactions, which are propagated appropriately across threads, and maintained as the nesting level changes. This property complements the ambient Transaction.Current property (making up for the absence of a Transaction.Parent). Our SSTM components are aware of this extension; it is public and usable by future nesting-aware RMs as well.

4.4 SibEx

The SibEx component implements the xfork API and coordinates the execution of sibling transactions. It may be used without TxStore, provided that the programmer has some other (nesting-aware) resource manager to use. The primary tasks of the SibEx component include the following:

- Scheduling the work for each fork to execute on some thread.
- Creating the sibling nested transactions.
- Invoking the fork procedure to execute within the proper context.
- Enforcing the semantics of each sibling form, which may include re-trying aborted transactions, killing unnecessary transactions, or coordinating the commit of multiple sibling transactions.

[1] The single minor exception will be noted in Section 5.4.3.

[2] System.Transactions also provides an ISinglePhaseNotification interface, which allows enlisters to avoid the prepare phase when they are the only resource manager in the transaction. However, our prototype does not currently implement this optimization. Since our two components are neither integrated with the CLR nor with each other, they would appear as two separate RMs and thus render the optimization unusable.

- Stalling the parent transaction/thread that calls xfork until the sibling transactions complete.

The implementation of each step is described in Section 5.

4.5 TxStore

The TxStore component provides basic software transactional memory functionality with support for parallel nested transactions. We built this component because the CLR does not provide built-in support for transactional memory. TxStore can be used even without the xfork API provided by the SibEx component. However, without the xfork API, coordinated sibling transactions are not available to the programmer. We configure some of our experiments to not use the xfork API as a baseline to measure the additional benefits of sibling transactions.

TxStore differs from previous STM designs due to requirements from the runtime environment and SibEx. The key differences are:

- TxStore is a generic store. TxStore operates at the granularity of objects, not memory bytes. The objects themselves may be integers, booleans (or any other built-in primitive types), *as well as* user-defined classes and structs.
- TxStore is a resource manager (from the System.Transactions point of view).
- TxStore supports true closed-nested transactions, as well as concurrent access by nested transactions of the same parent.

The detailed design of TxStore is presented in Section 6.

5. SibEx: Design and Implementation

The Sibling Executive (SibEx) is the component responsible for executing and coordinating sibling transactions. It contains the logic that transforms regular concurrent nested transactions into sibling transactions with the desired semantics. The SibEx assumes that the system supports concurrent nested transactions. As noted in the previous section, we were able to add this behavior to the CLR, and both SibEx and the TxStore are aware of our extensions. In Section 4.4, we gave a high-level view of the tasks that the SibEx must perform to execute an xfork request. This section discusses how each of those tasks is carried out.

5.1 Scheduling work

Design The first task is to schedule execution of different forks across a set of threads. The parent thread (which calls xfork) can also execute work items.

Implementation Our prototype uses the CLR's built in ThreadPool, which maintains a set of worker threads managed according to workload. Xfork packs the relevant state for each fork (forkProc, forkNum, data, SibExCoordinator, and the parent transaction) into a context object, and requests execution on a thread using ThreadPool's QueueUserWorkItem method. This is done once for each fork. We rely on the ThreadPool to determine the optimal number of threads to use, given hardware resources and the workloads of the application and system. The ThreadPool calls back into SibEx using a provided callback function, which was passed in to QueueUserWorkItem.

5.2 Creating sibling transactions

Design Once a fork is executed on a separate thread, the system must create a sibling transaction before invoking the forkProc. The sibling transaction must be a child of the parent thread's transaction.

Implementation The SibEx callback (invoked by the ThreadPool) creates a transaction using the TransactionScope mechanism. At a high level, this process is similar to the atomic keyword in existing STM proposals. The created transaction looks like a regular top-level transaction to the system. However, we ensure that the parent transaction, passed through the context object, is made available to resource managers by adding it in our thread-local parents list.

5.3 Invoking the fork procedure

Design The forkProc is then invoked within the context of the sibling transaction. The forkProc terminates either with a return code (success/failure) or an exception. In general, exceptions will be treated as an abort, and are functionally equivalent to the forkProc returning false. Depending on the type of sibling transaction, aborting a sibling transaction does not necessarily result in an xfork operation failure.

Implementation In the CLR, function pointers are represented by objects of type System.Delegate. The SibEx callback synchronously invokes the user-provided delegates, which execute within the transactional context. We define two new exceptions, TransactionRetryException and TransactionDoomedException, which have the obvious semantics when thrown by system components or the forkProc.

5.4 Enforcing the semantics of each sibling form

We previously defined the semantics of each of the sibling transaction forms (AND, OR, XOR). Since each form requires different logic, we implement three subclasses of an abstract class, SibExCoordinator, to perform the coordination specific to each of the forms. The SibExCoordinator class is used by the xfork method and the various threads executing the sibling transactions. The appropriate type of SibExCoordinator is created for each instance of xfork invocation, and passed to all the thread-pool threads as part of the work-item context object. Our design allows custom policies to be defined (as other implementations of the abstract class), which is important for enabling future research. We discuss the internals of the three provided SibExCoordinator implementations and how they interact with executing sibling transactions.

5.4.1 SibExCoordinator-OR

Design The simplest of the SibExCoordinator implementations is the one that handles OR-form invocations of xfork. This form is most similar to regular parallel nested subtransactions, the only notable difference being that the xfork implicitly performs a join before returning control to the parent transaction. The sibling transactions are independent, with each fork completing (commit or abort) without influencing the outcome of other transactions. The SibExCoordinator is informed by the fork threads whenever forkProc execution begins, ends, or throws an exception. The SibExCoordinator-OR attempts to retry any aborts caused by conflicts. Finally, when all forks have completed execution, the parent is signalled that the fork is over. A bitmask of the succeeding forks can be returned from the xfork for the programmer's use.

Implementation The SibExCoodinator-OR must retry transactions that abort due to conflicts. Here we encounter an important distinction between System.Transactions and traditional STM transactions: the former do not automatically retry. CLR transactions do not retry because they are designed in the database-style, not the TM style. The SibExCoordinators decide whether to re-execute the forkProc in a new transaction based on the result of the previous transaction execution. The transaction executions results in a commit (with success or failure return codes), an abort, or an exception.

To know whether a transaction aborted or committed, the SibExCoordinator must subscribe to the outcome-notification using the Transaction.TransactionCompleted event of each sibling transaction created. Once the SibExCoordinator determines whether a given fork has committed or aborted, it decides if a restart is appropriate. Since restarted transactions manifest as entirely new transactions, the SibExCoordinator is given a chance to subscribe to their outcome notification before forkProc execution begins.

5.4.2 SibExCoordinator-AND

Design The SibExCoordinator-AND has one key difference from the SibExCoordinator-OR: it must influence, not simply learn, the outcome of each transaction. The semantics of the AND form mandate that none of the sibling transactions commit unless they all commit. They are similar in some respects to a traditional fork/join barrier.

Note that the xfork API reports failures to the parent thread only when it can guarantee that *none* of the transactions have committed. Execution never returns to the caller in a state where some forks have committed while others have aborted. Thus, the parent transaction always knows exactly what the program state is after the xfork and can continue execution appropriately.

Implementation The SibExCoordinator-AND enlists in every sibling transaction before the forkProc is called. By enlisting, it participates in the two-phase commit process, and thus can influence the outcome of each transaction. It does not allow any transaction to commit unless they all reach the prepare phase. If this occurs, then xfork returns a successful prepare to all of the transactions. If any fork fails, xfork aborts all sibling transactions.

The possibility remains that one or more of the transactions will still abort after being prepared by SibEx (due to decisions of other resource or transaction managers). As mentioned previously, in situations where the implementation is unable to meet the atomicity guarantee of the xfork API, it forces a restart of the calling (parent) transaction.

5.4.3 SibExCoordinator-XOR

Design The SibExCoordinator-XOR is similar to the AND coordinator in that it must influence the outcome of a transaction. It has two additional requirements. First, only one of the sibling transactions is allowed to commit. Second, if any transaction commits, then any other sibling forks (and transactions) still executing are useless and no longer needed, and should be aborted/terminated. This termination can be accomplished by actually tearing down the executing threads, or by some other mechanism to preempt the execution of the forkProcs on those threads.

We maintain a state machine to coordinate the states of the different sibling transactions. If no transaction has yet committed, the coordinator only allows one transaction to proceed past the prepare phase, stalling any others that reach that phase. If the outstanding prepare request commits, then all other prepare requests are aborted and all future transactions must abort. However, if the outstanding prepare does not commit, then another prepare request is allowed to attempt a commit.

Implementation Terminating unneeded forks and aborting their associated sibling transactions involves two steps. First, the actual transactions are aborted by failing all prepare requests that arrive after a sibling has committed. This is accomplished by virtue of the SibEx coordinator being enlisted in the transaction (and thus having a vote). As for terminating the execution of the fork, it was impracticable to use the Thread.Abort API to actually terminate the threads. Not only is the API discouraged (and very slow), but the threads we use are owned by the ThreadPool (and thread-replacement policies introduce additional delays for subsequent work items). Instead, we terminate the forkProc's execution, without destroying the actual thread-pool thread.

This is done by having the coordinator call into the resource-manager (the TxStore), and informing it of the set of transactions it knows it will abort (i.e. those that are doomed). The TxStore then throws a TransactionDoomed exception when a doomed transaction attempts to perform an operation on it. This exception is ultimately caught and swallowed by the SibEx callback executing on the thread-pool thread. This method allows quick cleanup (assuming that the siblings will frequently be invoking the TxStore). An implementation of sibling transactions that is more integrated in the CLR runtime environment would not have to make this assumption, as it would be able to hijack control of any thread at more opportunities.

```
class TxStore
{
  object Read(int address);
  void Write(int address, object obj);
  int Allocate()
  void Free(int address)
}
```

Figure 6. TxStore API

5.5 Stalling the parent transaction

Design The xfork call must know when it is safe to return control to the calling thread (parent transaction). As discussed above, the SibExCoordinator managing an invocation of xfork knows when it is safe to return, as well as the operation's result. This event does not necessarily mean that all sibling transactions have finished execution on the other threads, only that the necessary committed ones have completed. Other transactions in the XOR form may still be executing, but ultimately do not affect the execution of the parent transaction because the coordinator ensures that they will abort.

Implementation Our parent thread waits on a ManualResetEvent provided by the SibExCoordinator, which is set when it is safe to return to the parent transaction.

6. TxStore: Design

The TxStore is a nesting-aware transactional object store used by our prototype. This section describes its API, high-level design and key data structures and operations.

6.1 API

The TxStore exposes a public API, which may be used directly by the programmer, or, when tightly integrated with a language or runtime, may be indirectly accessed on the programmer's behalf. The API, shown in Figure 6, has transactional semantics and is similar to a dynamically-addressed dictionary. Objects that are written must implement the System.ICloneable interface (if they are not value-copied primitives), which allows the TxStore to make a clone for storage and to clone objects in response to read requests. Cloning objects ensures that active transactions do not directly manipulate the stored object, but rather a clone in their working set.

Unlike most previous STMs, which operate at the word or class level, TxStore is a generic (i.e. based on System.Object) store that can be used for both. TxStore leverages the CLR's ability to treat any variable as an object (including primitive types, which also derive from System.Objects), relying on the transparent boxing and unboxing of CLR objects. TxStore does not define the granularity of the objects in the programmer-visible method. The language designer decides the mapping of programmer-visible classes to

TxStore-managed objects, and the TxStore implementation handles both. For example, an object with five member fields can be stored as one object within the TxStore, or each member of the object can be stored as a separate object within the TxStore. The finer granularity can be used to reduce conflicts, such as when different members are updated concurrently by different transactions. Our prototype assumes that the entire object is treated as a single unit.

6.2 Design

The TxStore design can be seen as an extension of TL2 [5], adapted to support parallel nested transactions. As with TL2, the design accepts serializable executions (equivalent to 2-phase locking) with conflict detection based on time-stamps. TL2 maintains a single write timestamp per transaction, and only maintains read versions for items in the working set. However, TxStore maintains write versions for each item in the working set. The relation of main memory to TL2 transactions is similar to that of parent transaction working sets to parallel nested transactions. TxStore's use of per-item write versions allows parallel nested transactions to modify different subsets of a single parent's working set.

TxStore uses lazy conflict detection, and the read set is validated at commit time against a parent transaction's copy or the committed item. Isolation is provided by locking the write set at prepare time, and holding these locks until commit or abort. Deadlocks cannot arise if the TxStore is being used alone, as conceptually all locks are acquired in a known order. However, when used with coordinated sibling transactions, deadlocks can arise in certain cases. We deal with any deadlocks by using a simple timeout mechanism. All internal data structures (including working sets) need to be protected against concurrent access by children transactions. Our implementation uses locking for simplicity (though we anticipate hardware assistance could be particularly helpful in this area).

6.3 Data structures

Internally, the TxStore maintains two maps. The first is a map of committed objects (StoredObjects), indexed by address (see Figure 7). The second map is of active transactions that are accessing the TxStore. Each transaction, whether top-level or nested, is tracked using an internal RMTransaction object. The nesting relationships between transactions is tracked by the TxStore, using a pointer to the parent RMTransaction. Beyond the status field, the most critical member of RMTransaction objects is the working-set, a hashtable of AccessedObjects indexed by address. Figure 7 shows the fields of the AccessedObject class.

6.4 TxStore operations

We present the high-level logic for a transaction performing a read and write operation (assuming a previously created object is being accessed). We also show the steps performed on transaction commit and abort. We omit the allocate/free APIs, as well as the edge cases where the data structures may concurrently be accessed while being allocated or freed. These cases are all available in the source code, which will be made publicly available.

TxStore.Read:

1. Enlist in the transaction if necessary, creating the RMTransaction internal object.
2. Look in the RMTransaction.working-set for the address. If found, then return a clone of AccessedObject.realObj.
3. If not found, search the chain of parent RMTransactions for the first working-set that contains the requested address. If found, propagate it down to all working sets in the chain, including that of the current transaction, while setting the readFlag for each AccessedObject created. Each newly created AccessedObject's readVersionNum field is copied from the changeVersionNum of the AccessedObject found at the top of the chain. To protect against races from other concurrent children, each RMTransaction object has a lock that is acquired while accessing/modifying its working set.
4. If no parent has accessed the object, look up the address in the StoredObjects map and copy the realObj into the current RMTransaction's working set. The StoredObjects map is synchronized for the duration of this lookup (currently using a lock though finer synchronization is possible). We set the readFlag for the created AccessedObject, and its readVersionNum is copied from the StoredObject's versionNum.

```
class StoredObject
{
  object realObj;
  int versionNum;
  bool deleted;
}
class AccessedObject
{
 object realObj;
 int readVersionNum;
 int changeVersionNum;
 bool readFlag;
 bool writtenFlag;
 bool addedFlag;
 bool deletedFlag;
 StoredObject lockedObject;
}
```

Figure 7. TxStore internal data structure. The StoredObject is for committed objects, while the AccessedObject is maintained by active transactions in their working sets.

TxStore.Write:

1. Enlist in the transaction if necessary, creating RMTransaction internal object.
2. If the RMTransaction working-set does not contain an AccessedObject for the requested address, create one.
3. Write the object into the existing or created AccessedObject.realObj field, and set the writeFlag. A newly allocated version-number is written into the changeVersionNum field.

RMTransaction.Prepare:

1. Lock StoredObject map.
2. For each AccessedObject in the RMTransaction's working-set:
   (a) Lock the StoredObject, and set the AccessedObject's lockedObject field to point to the StoredObject.
   (b) If the AccessedObject readFlag is set, ensure that the readVersion is equal to the nearest parent's AccessedObject (changeVersionNum, otherwise readVersionNum), or the StoredObject's versionNum. If not, release all locks, and fail the Prepare request.
3. Unlock the StoredObject map.

RMTransaction.Rollback:

1. For each AccessedObject in the RMTransaction's working set, release the lockedObject if it is not-null. This is done without taking the StoredObject map lock.

RMTransaction.Commit:

1. Lock the RMTransaction object to prevent any races from concurrent children.
2. For each AccessedObject in the RMTransaction's working-set:
   (a) If the transaction is a top-level one and the AccessedObject writeFlag is set, copy the realObj and the changeVersionNum to the StoredObject's realObj and versionNum fields.
   (b) If the transaction is nested and the parent does not have an AccessedObject for the same address, transfer the entire AccessedObject to the parent's working set.
   (c) If the transaction is nested and the parent has a corresponding AccessedObject, merge the current AccessedObject with the parent's AccessedObject. If the transaction has only read (readFlag set), then no merging needs to occur because the prepare phase ensured that the version number of the child is identical to the parent regardless of whether the parent has read or written. If the transaction has written (writeFlag set), the changeVersionNum and realObj are propagated to the parent, whose writeFlag is also set.
   (d) Unlock the StoredObject, which is in the AccessedObject's lockedObject field.
3. Unlock the RMTransaction object

7. Evaluation

This section describes our evaluation of the SSTM prototype, the benchmarks used, and the speedup results compared to our base-line STM configuration.

7.1 Prototype and evaluation goals

Our prototype is not fully optimized. The goals of this evaluation is to get a sense of the efficacy of the system and the overheads involved. We thus use a focused set of three benchmarks, which cover the three forms of coordination.

7.2 Software and hardware setup

SSTM is implemented on top of the Microsoft .NET Framework version 2.0.50727. The system consists of 2,345 lines of C# code, including about 600 lines of benchmark code. We make extensive use of the base class libraries (collection classes, etc.) and synchronization primitives (WaitEvent, Monitor, etc.), and have not bothered with many of the usual optimizations (e.g. reader-writer locks, specialized data structures, avoiding kernel calls caused by wait-objects, etc.). Our experiments are run on an Intel Core2 Quad CPU running at 2.66 Ghz, with 4GB of RAM. The OS is Microsoft Windows Vista Ultimate (32-bit), with SP1.

7.3 Benchmarks

We evaluate SSTM with three benchmarks, each corresponding to one of the forms of sibling transactions. They allow us to evaluate xfork overhead compared to benefit. Each benchmark is written with the serial transactional code (the base-line version), and then modified to use xfork to produce 2 and 4-core SSTM versions.

7.3.1 SearchList

The SearchList benchmark has a transaction that finds a random element within a doubly linked list. We compare the base version, which performs a forward traversal of a linked list, with the xfork version. The xfork version uses the XOR form to search the list. As soon as one of the siblings has found the node, the others are destroyed. The 2-core version performs both forward and reverse traversals, while the 4-core version performs two additional traversals starting from the middle of the linked list (maintained by the structure).

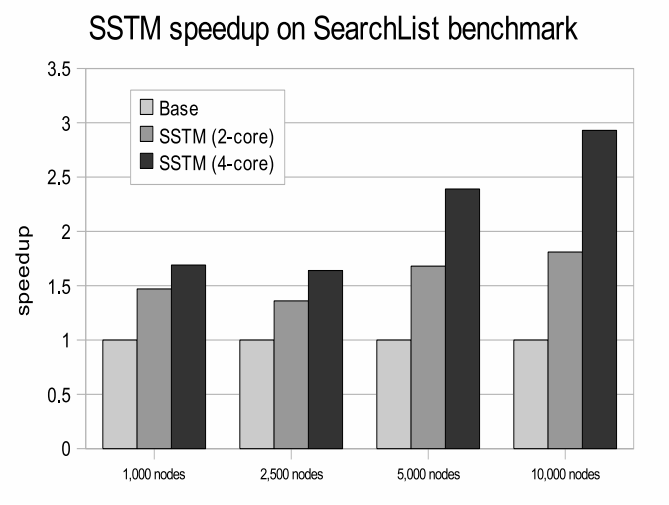

Figure 8. Speedup of SSTM on the SearchList benchmark, for different length lists.

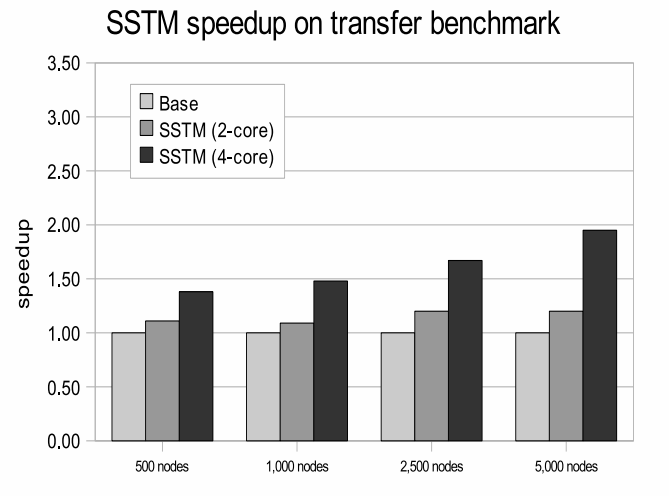

Figure 9. Speedup of SSTM on the Transfer benchmark, for different length lists.

We vary the number of nodes in the list from 1,000 to 10,000 nodes. The results are shown in Figure 8. Sibling transactions are able to provide a significant speedup over the regular linked list traversal code, with minimal programmer effort.

7.3.2 Transfer

The transfer benchmark exercises the AND form of sibling transactions. It is a debit-credit operation that is a simplified form of the transfer example presented earlier in this paper (only two lists are involved, not three). The accounts exist in two separate lists, and accounts are randomly selected. We incorporate a fixed think time to model the work involved when the appropriate account nodes are actually found. We compare the base transaction code, where the debit is done followed by the credit, with an xfork version, where the debit and credit are done concurrently. The debit fork returns a failure code if the balance is insufficient. The 2-core benchmark uses regular linked list traversal in each of the credit and debit forks.

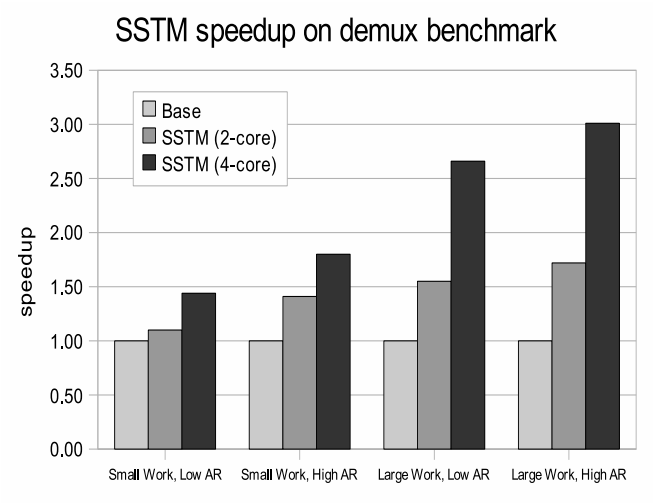

Figure 10. Speedup of SSTM on the demux benchmark. Low AR models no conflicts, while High AR models 12% conflict rate. Small Work uses a think-time of 10kC, while Large Work uses a think-time of 50kC.

The 4-core benchmark adds a speculative search of each list, in effect nesting an invocation of the SearchList benchmark. The results are shown in Figure 9. The speedups are less pronounced compared to SearchList because of the additional overhead of AND coordination (which involves enlistments) compared to OR. The 4-core results demonstrate that multiple levels of sibling transactions are practical, and further improves performance in the presence of the additional cores.

7.3.3 Demux

The demux benchmark involves a transaction that dequeues a work item, processes it, and then attempts to enqueue two or more subsequent work items to further queues. All operations are done within a single transaction so that any failures leave the system in a consistent state. Enqueue operations can sometimes fail, for example when conflicts occur with other top-level transactions, or if the queue is temporarily full. The benchmark models this by failing enqueues at a specified probability. We also model the work done before the transaction enqueues by a variable think time (measured in kilo-Cycles). We execute 20,000 transactions over the duration of the run. We modified the benchmark by performing an xfork that performs the enqueue operations using the OR form, in both 2 and 4 core versions.

The results are shown in Figure 10 as speedups of the SSTM version to the baseline code. The SSTM versions are able to speedup the baseline version on 2 cores, and scales noticeably on 4 cores. Note that as the abort rate increases, the relative speedup of SSTM over the baseline also increases. This increase is a benefit of nested transactions in general, since less work is wasted when an enqueue operation fails, since only that action (done within a sibling transaction) is retried, not the work done by the parent transaction. The user of xfork automatically gets the benefits of nested transactions. By the same token, increasing the amount of work (think time) in the parent transaction leads to better scaling.

7.4 Benchmark results summary

These benchmarks show that for a very small amount of programming effort, we can make sequential transactions run even faster by leveraging additional cores while retaining the benefits of transactions. We demonstrate a speedup of up to $1.87\times$ (on two cores) and $3.12\times$ (on four cores) on the demux benchmark. On the more substantial transfer benchmark (with nested xforks), we see a speedup of $1.2\times$ (on two cores) and $1.95\times$ (on four cores). These results, obtained on our unoptimized C# implementation, are encouraging, and show that further research and experience with xfork in software and hardware is warranted, so as to understand better its strengths and limitations.

8. Related work

This section discusses the most relevant related work.

Nested transactions Nested transactions were introduced long before transactional memory [20, 4], and were used in early distributed systems such as Argus [15] and Camelot [7]. Nested transactions are only one type of advanced transaction model. Others are surveyed by Gray[8] (ch. 4) and Weikum [24]. Application-specific transaction models tend to be even more complex (and less general). These are surveyed by Elmagarmid [6] and include cooperating transactions, used by some CAD applications, which interact to pass ownership of resources.

Transactional memory Larus and Rajwar provide a comprehensive reference of software and hardware transactional memory systems as of summer 2006 [14]. Transactional memory systems initially did not implement any real nesting, making do with a flat transaction model. More recently, however, there have been several proposals that incorporate true nesting into transactional memory, both in hardware [18, 16] and software [11, 21]. Moss and Hosking sketch a reference model of both open and closed nesting, as well as a simpler model, which supports a single thread of execution only, called linear nesting [19].

Language extensions that are related include orElse [10] and t_for [9] constructs, the latter focusing on the well-known issue of loop parallelization. Fortress [2] allows (independent) parallel nested atomic blocks, as does XCilk [1].

Concurrency in the presence of conflicts Recent work on allowing TM systems to accept more schedules [3, 22] allow transactions to commit that would otherwise restart. These approaches are orthogonal to sibling transactions and can be used alongside them. Sibling transactions create more concurrent transactions, so approaches that reduce aborts caused by conflicting transactions should improve the performance of sibling transactions. Approaches such as dependence-aware transactions would have to be extended to take into account inter-transaction dependences, and treat them differently than inter-sibling dependences.

STM programming model extensions Many proposals exist for changes to the TM programming model to increase performance. Among these are early release [23], escape actions [25], and Galois classes [13] and boosting [12]. These proposals often are a good fit in limited applications, and usually require the programmer to be very careful to avoid subtle consistency and isolation problems. By contrast, coordinated sibling transactions are more general, straightforward to use and retain the full protection of closed-nested transactions.

9. Conclusion

The xfork API allows programmers to easily express inherent concurrency within their atomic sections, while retaining the simplicity of transactions. Coordinated sibling transactions make parallel nested transactions a commodity. We presented the design of SSTM, our prototype system built on the CLR. The evaluation shows the efficacy and potential of this model.

References

[1] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *PPoPP*, 2008.

[2] E. Allen, D. Chase, J. Hllett, V. Luchango, J. Maessen, S. Ryu, and S. Tobin-Hochstadt. *The fortress Language specification 1.0*, 2007.

[3] U. Aydonat and T. Abdelrahman. Serializability of transactions in software transactional memory. In *TRANSACT*, 2008.

[4] C. Beeri, P. Bernstein, N. Goodman, M. Lai, and D. Shasha. A concurrency control theory for nested transactions. In *PODS*, 1983.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, 2006.

[6] A. Elmagarmid. *Database Transactional Models for Advanced Applications*. Morgan Kaufmann, 1992.

[7] J. Eppinger, L. Mummert, and A. Spector. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.

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

[9] L. Hammond, B. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency. In *ASPLOS*, October 2004.

[10] T. Harris, M. Herlihy, S. Marlow, and S. Peyton-Jones. Composable memory transactions. In *PPoPP*, Jun 2005.

[11] T. Harris and S. Stipic. Abstract nested transactions. In *TRANSACT*, 2007.

[12] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*, 2008.

[13] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[14] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[15] B. Liskov. Distributed programming in argus. In *CACM 31(3)*, 1988.

[16] A. McDonald, J. Chung, B. Carlstrom, C. C.M., H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *ISCA*, Jun 2006.

[17] Microsoft Corp. *.NET Framework Developer Center*, 2008. `http://msdn.microsoft.com/en-us/netframework`.

[18] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS*, 2006.

[19] E. Moss and T. Hosking. Nested transactional memory: Model and preliminary architecture sketches. In *SCOOL*, 2005.

[20] J. Moss. Nested transactions: An approach to reliable computing. In *MIT LCS-TR-260*, 1981.

[21] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP*, 2007.

[22] H. Ramadan, C. Rossbach, and E. Witchel. Dependence-aware transactions for increased concurrency. In *MICRO-41*, 2008.

[23] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads*, Jun 2006.

[24] G. Weikum and H. Schek. *Concepts and Applications of Multilevel Transactions and Open Nested Transactions*. Morgan Kaufmann, 1992.

[25] C. Zilles and L. Baugh. Extending hardware transactional memory. In *TRANSACT*, Jun 2006.