Polymorphic Contention Management

Rachid Guerraoui[1], Maurice Herlihy[2⋆], and Bastian Pochon[1]

[1] School of Computer and Communication Sciences, EPFL
[2] Brown University and Microsoft Research Cambridge

Abstract. In software transactional memory (STM) systems, a contention manager resolves conflicts among transactions accessing the same memory locations. Whereas atomicity and serializability of the transactions are guaranteed at all times, the contention manager is of crucial importance for guaranteeing that the system as a whole makes progress.
A number of different contention management policies have been proposed and evaluated in the recent literature. An empirical evaluation of these policies leads to the striking result that there seems to be no "universal" contention manager that works best under all reasonable circumstances. Instead, transaction throughput can vary dramatically depending on factors such as transaction length, data access patterns, the length of contended vs. uncontended phases, and so on.
This paper proposes *polymorphic contention management*, a structure that allows contention managers to vary not just across workloads, but across concurrent transactions in a single workload, and even across different phases of a single transaction. The ability to mix contention managers or to change them on-the-fly provides performance benefits, but also poses number of questions concerning how a contention manager of a given class can interact in a useful way with contention managers of different, possibly unknown classes. We address these questions by classifying contention managers in a hierarchy, based on the cost associated with each contention manager, and present a general algorithm to handle conflict between contention managers from different classes. We describe how our polymorphic contention management structure is smoothly integrated with nested transactions in the SXM library.

1 Introduction

Because it is getting harder and harder to make processors run faster, chip manufacturers are focusing on *multicore* architectures, in which multiple processors (cores) communicate directly through shared hardware caches [6]. The next generation of processors will provide increased concurrency instead of increased clock speed, and programming languages and APIs will need to exploit this increased parallelism.

The limitations of conventional synchronization techniques, based on locks and condition variables [1] are well-known [11, 10]. Coarse-grained locks, which protect relatively large amounts of data, simply do not scale. Threads block one another even when they do not really interfere, and the lock itself becomes a source of memory contention. Fine-grained locks are more scalable, but they are difficult to use effectively and correctly. In particular, they introduce substantial software engineering problems, as the conventions associating locks with objects become more complex and error-prone. Locks also cause vulnerability to thread failures and delays: if a thread holding a lock is delayed by a page fault, or context switch, other running threads may be blocked.

An alternative to locking is to synchronize by light-weight in-memory *transactions*, an approach called *transactional memory*. A transaction [2] is a finite sequence of memory reads and writes executed by a single thread. Transactions are atomic [19]: each transaction either commits (it takes effect) or aborts (its effects are discarded). Transactions are serializable [14]: they appear to take effect in a one-at-a-time order. (Unlike database transactions, we are not concerned here with backing up changes to non-volatile memory.)

Software transactional memory (STM) systems have been the focus of much recent research [7, 10, 8, 16, 17]. Most of these systems guarantee a relatively weak progress property called *freedom from obstruction* [9]: if a transaction runs long enough without overlapping a conflicting transaction, then it will commit. Obstruction-freedom does not rule out livelock or starvation, so stronger progress properties are typically provided "out-of-band" by a user-provided module called a *contention manager*. Roughly

⋆ Supported by NSF grant 0410042 and by grants from Intel Corporation and Sun Microsystems.

speaking, if transaction A is about to take a step that will cause a synchronization conflict with transaction B, then A consults its contention manager to decide whether to proceed, thus causing B to abort, or else to back off for a bounded duration, giving B a chance to finish.

Contention managers affect liveness, not safety. The most natural way to evaluate a contention manager is by its *throughput*, the number of transactions committed per unit of time. A bad transaction manager provides low throughput, but cannot produce unsafe results (except perhaps by throwing unexpected exceptions).

A number of different contention management policies have been proposed [5, 10, 16, 17]. In contrast with a recent publication [17], a striking result of our evaluation (discussed in more details below) is that there seems to be no "universal" contention manager that works best under all reasonable circumstances. Instead, transaction throughput can vary dramatically depending on factors such as transaction length, data access patterns, length of contended vs. uncontended phases, and so on. We expect this uncertainty to be even more drastic in large scale concrete applications.

Figure 1 illustrates how contention manager performance depends on context. The figure features contention managers that have recently appeared in the literature (we give more details on these later in the paper). The two scenarios illustrated differ in the contention pattern among conflicting transactions. Both scenarios use a red-black tree data structure in which a number of threads insert and remove elements. The number of transactions committed within a constant time period of one second, under various contention management policies, is depicted, with respect to a number of threads ranging from 1 to 32. The scenario on the left reduces contention by making each thread executes a time-delay loop at the end of every transaction. The scenario on the right exhibits a contention-intensive scenario, in which threads continuously insert and remove elements from the red-black tree. Elements are taken from a small set of 256 integers to force contention to happen within the tree. Benchmarks were run on a 4-processor Intel Xeon machine with hyperthreading turned on.

When situations such as those of Figure 1 *simultaneously* appear within a single application, the application benefits from associating distinct contention managers to different groups of transactions, according to the situation encountered by each particular group. The diversity of contention managers within a single application may also be motivated by the change, over the lifetime of the application, of parameters affecting the throughput of committed transactions, for instance the number of threads or the number of tables in a database. In this case, running transactions with a default contention manager from some time on, may lead to conflicts with transactions that are still running with the previous default contention manager and have not yet committed.

We propose *polymorphic contention management*, a structure that allows contention managers to vary not just across workloads, but across concurrent transactions in a single workload, and even across different phases of a single transaction. The ability to mix contention managers or to change them on-the-fly provides performance benefits, but also poses number of questions concerning how one contention manager of a given class[3] can interact in a useful way with contention managers of different, possibly unknown classes. We introduce a hierarchy of contention manager classes, based on the cost associated with each contention manager class, and identify general groups of contention manager classes. We present a general algorithm to handle contention between contention managers from different classes.

Our polymorphic contention management structure is presented in the context of SXM, a new software transactional memory library which we implemented in C#. SXM supports distinct contention management policies at the level of individual, possibly nested [13], transactions. We associate transactions with methods in SXM, which makes it natural to isolate nested transactions from parents, and concurrent transactions from one another. Our polymorphic scheme promotes a flexible programming style where the contention manager of a nested transaction can be interactive: the thread of control is returned to the application after the transaction is aborted a certain number of times. This allows the application for possibly changing at runtime the contention manager of the transaction (e.g., if the number of threads increases). The full code of SXM is available on the web for further experimentation [15].

The remainder of the paper is organized as follows. Section 2 gives an overview of elements of SXM that are needed to describe our polymorphic contention management structure. Section 3 explains in more details contention management in SXM and compares different contention managers. Section 4 discusses the mixing of contention managers. Section 5 describes how to associate a contention manager with a

[3] Throughout the paper, the notion of "contention manager class" is to be taken in the object-oriented sense, i.e., a set of instances implementing the same contention manager policy.

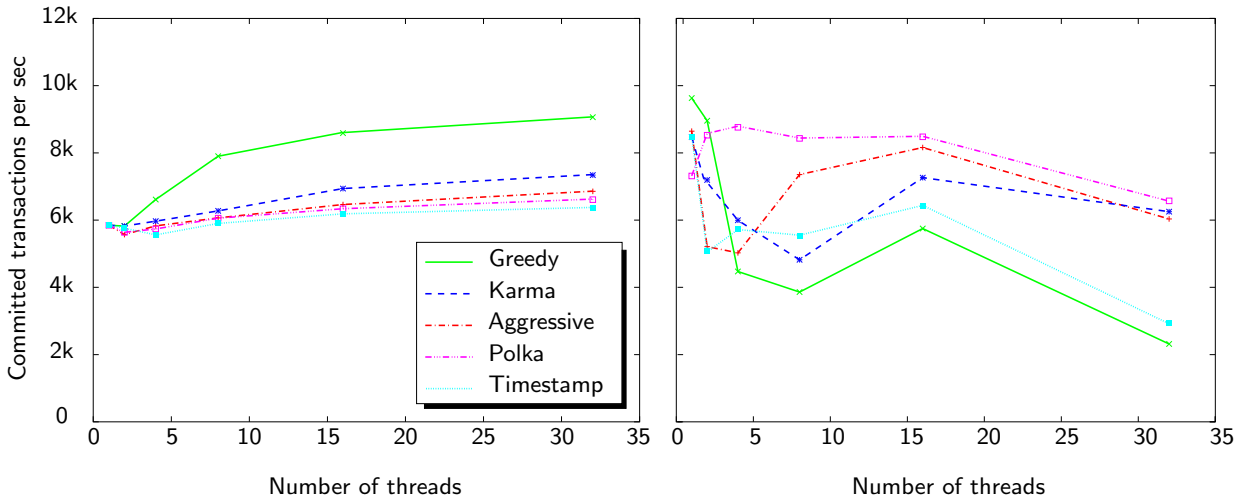


Fig. 1. Comparison of various contention management policies under low (left) and high (right) contention scenarios

transaction in SXM. Section 6 presents the implementation of SXM in C#, including a postprocessor approach to declaring transactions. Section 7 discusses related work.

2 Overview of SXM

Before describing our polymorphic structure, we provide here a short overview of our transaction model, illustrated in Figure 2, and an example of a program using our SXM library. In Section 6, we will give more details about the implementation of SXM and the polymorphic contention management scheme in C#.

2.1 Transactions

We consider a system made of n threads. Each thread executes independently of other threads, and follows a program assigned to it. Beside normal code, a thread may execute *transactions* [2]. A transaction is a basic unit of computation that appears to take place atomically [19] to every other transaction (thread). When a thread finishes a transaction, the transaction may either *commit*, and every modification performed during the transaction instantaneously takes place, or *abort*, in which case no modification is effectively performed.

We consider an object-based memory model. In this model, threads share objects. Within a transaction, a thread accesses *transactional objects*. A transactional object is an object supporting additional methods, for being accessed from within a transaction. Roughly speaking, every transactional object supports a *clone* operation. When a thread executes within a transaction, the thread works on a *copy* of the transactional object, obtained using the clone method. When the thread obtains a copy of the transactional object to work on, we say the thread *acquires* the transactional object. Upon completion, the transaction atomically commits every modification done on every copy acquired, to the original transactional object. If the transaction fails to commit, the thread may restart the transaction.

2.2 Contention managers

A transaction T_a may fail to commit because another transaction T_b has accessed the same transactional object O, invalidating the copy T_a has obtained. When T_b acquires O, T_b detects a conflict with T_a. At this point however, it is not clear whether T_b, which we call the *attacking* transaction, should abort T_a, which we call the *victim* transaction, or whether T_b should just wait and give more time to T_a to finish. It is possible that T_a never commits if it is always aborted by other transactions. Hence the choice of mediating conflicts among transactions with a *contention manager*.

More precisely, a contention manager instance is associated with each transaction. In case of a conflict, the contention manager of the attacking transaction decides, possibly based on the computation of

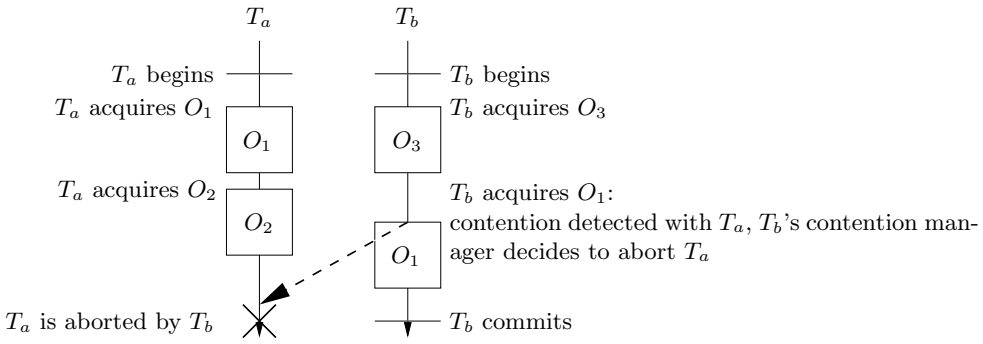


Fig. 2. T_b is attacking victim transaction T_a

the transactions, which of the attacking or the victim transaction should abort. On the one hand, a transaction that has aborted many times should be given a change to commit, and thus should not be aborted too easily. On the other hand, a transaction may be blocked, e.g. waiting for swapped-out data to be swapped in, and aborting the blocked transaction in favor of others may lead to better throughput. Clearly, the contention manager is crucial to the performance of a STM system.

By eventually aborting any conflicting transaction if called sufficiently many times, a contention manager may easily ensure *freedom from obstruction* [9], a relatively weak liveness guarantee. More sophisticated contention management policies ensure stronger liveness guarantees [5, 16].

2.3 Transactions and Contention Management in SXM

In SXM, the transaction invocation tree is mapped onto the method invocation tree, through a few programming conventions, following the current "library" approach of SXM. Section 6.3 describes an alternative postprocessor approach to declare transactions in SXM.

A transaction is explicitly constructed from a method and the transactional objects must be marked with the Atomic attribute and support the ICloneable interface. We assume here that a particular contention manager has been chosen and specified elsewhere, and we explain how this is effectively achieved in Section 3.

Figure 3 depicts a simple example of a counter in SXM. There are two classes, Application and Counter. The class Application represents the application class, and defines a method Increment, to be run as a transaction. The class Counter implements ICloneable and is marked with Atomic: hence instances of Counter are transactional objects, and may be accessed through the C# property inc, defined in Counter. (We use properties instead of methods because properties make an explicit distinction between get and set accesses, that we respectively associate with read and write accesses. This allows for distinguishing, within a transaction, objects that are accessed read-only, i.e. through a get property, or read-write, i.e. through set property).

The body of the Increment method consists in accessing a single transactional object, counter, instantiated from the class Counter, by invoking the increment property on it. The Main method declares a transaction corresponding to the Increment method, in three steps (line numbers refer to Figure 3):

1. Create first a delegate, representing the method to run as a transaction (line 6, second column). (A delegate is a C# feature that represents a kind of type-safe pointer on a method.)
2. Create a transaction, represented as an instance of SXMAction, from this delegate (line 8, second column).
3. Launch the transaction by invoking the Run method on the SXMAction instance (line 11, second column). (In a real application, running the transaction would obviously occur within a new thread.)

3 Specifying a Contention Manager

3.1 Contention Management Methods

Every transaction is associated with a particular contention manager, where the task of the contention manager is to resolve conflicts encountered with other transactions. The contention manager resolves

```
1: class Application
2:    Counter counter;          // Transactional object

3:    public Application()
4:      this.counter =
                ⟨obtain a fresh Counter instance⟩;

5:    // Increment will be run as transaction
6:    public void Increment()
7:      this.counter.increment;

8: static public void Main(string argv[ ])
9:    // Create the application
10:   Application application = new Application();

11:   // Create a delegate
12:   Delegate delegate = new
          SXMDelegate(application.Increment);

13:   // Create the transaction
14:   SXMAction incrementAction =
          SXMAction.Create(delegate);
15:   ...
16:   // Run the transaction
17:   incrementAction.Run();
```

```
1: // Class representing transactional objects
2: [Atomic]
3: class Counter : ICloneable
4:    private int balance;

5:    public Counter(int balance)
6:      this.balance = balance;

7:    public object Clone()
8:      return new Counter(this.balance);

9:    // Property that increments the balance
10:   property int increment
11:     set
12:       balance = balance + 1;
```

Fig. 3. Example of a counter in SXM

conflicts based on the computation performed by its associated transaction. In this sense, the contention manager is informed by its associated transaction of the evolution of the computation.

A contention manager exports *notification* and *feedback* methods: notification methods enable the transaction to inform the contention manager of its computation, whereas *feedback* methods enable the contention manager to inform the transaction of what to do in specific situations.

Notification methods include methods such as BeginTransaction, TransactionCommitted, TransactionAborted, as well as methods to indicate an attempt in acquiring a transactional object (OpenReadAttempt and OpenWriteAttempt), and success in acquiring a transactional object (OpenReadSucceeded and OpenWriteSucceeded).

A feedback method is invoked by a transaction on its own contention manager, in situations where the expertise of the contention manager is needed. We consider two feedback methods:

- The method ResolveConflict is invoked on the contention manager of a transaction whenever a conflict is detected with another transaction. Roughly speaking, the contention manager of the attacking transaction may decide in this case, according to its specific contention management policy (we give examples in the next section), whether to abort the victim transaction, or whether to send to sleep the attacking transaction and give more time to the victim transaction to finish.
- The method ShouldBegin is invoked on the contention manager of a transaction whenever the transaction (re)starts. This method returns whether the transaction should wait, or whether it may start. In a typical contention manager implementation, ShouldBegin blocks the transaction, based on its specific contention management policy, yielding in favor of other threads. The contention manager sends the transaction to sleep until it is appropriate for the transaction to start.

3.2 Examples of Contention Managers

Several contention managers have been defined in the literature [5, 16, 17]. The Aggressive contention manager systematically aborts the victim transaction. The Polite contention manager exponentially backs off for a fixed number of attempts, and eventually aborts the conflicting transaction. The Randomized contention manager aborts the victim transaction with some probability p, and waits with probability

```
1: static public void Main(string argv[ ])
2:     Application application = new Application();
3:     Delegate delegate = new SXMDelegate(application.Increment);
4:     SXMAction incrementAction = SXMAction.Create(delegate,typeof(Greedy));
5:     ...
6:     incrementAction.Run();
```

Fig. 4. Specifying a contention manager

$1-p$. Greedy and Timestamp contention managers associate a timestamp to a transaction when it is run for the first time. The idea is that an old transaction (one with a lower timestamp) has priority over a young one (one with a higher timestamp). In case of conflict, if the victim has a higher timestamp, it is aborted. With Greedy, if the victim transaction is waiting, then the attacking transaction aborts it; otherwise, the attacking transaction waits, until the victim either commits, aborts, or waits. With Timestamp, if the attacking transaction is not older than the victim, the attacking transaction then waits for a series of fixed intervals. After attempting half the number of intervals, the contention manager of the attacking transaction flags the victim as possibly defunct. After attempting the full number of intervals, if the victim has the defunct flag set, the contention manager of the attacking transaction aborts the victim; meanwhile, if the victim transaction performs any transaction-related operation, its contention manager resets the defunct flag. Karma and Polka contention managers increase the priority of a transaction whenever the transaction successfully acquires a transactional object. When two transactions are in conflict, the attacking transaction makes a number of attempts equal to the difference among priorities of both transaction, with a constant backoff between each attempt (Karma), or with an exponential random backoff between successive attempts (Polka). The Eruption contention manager also maintains the number of transactional objects successfully acquired, denoted *objs*, and gives to a transaction an initial priority of *zero*. In case of conflict, if the victim transaction has a higher priority than the attacking one, the contention manager of the attacking transaction adds *objs* to the priority of the victim transaction, and then sends the attacking transaction to sleep for an exponential random backoff. Otherwise, the contention manager aborts the victim transaction. The idea behind the Eruption contention manager is to increase the priority of the transaction behind which other transactions are waiting.

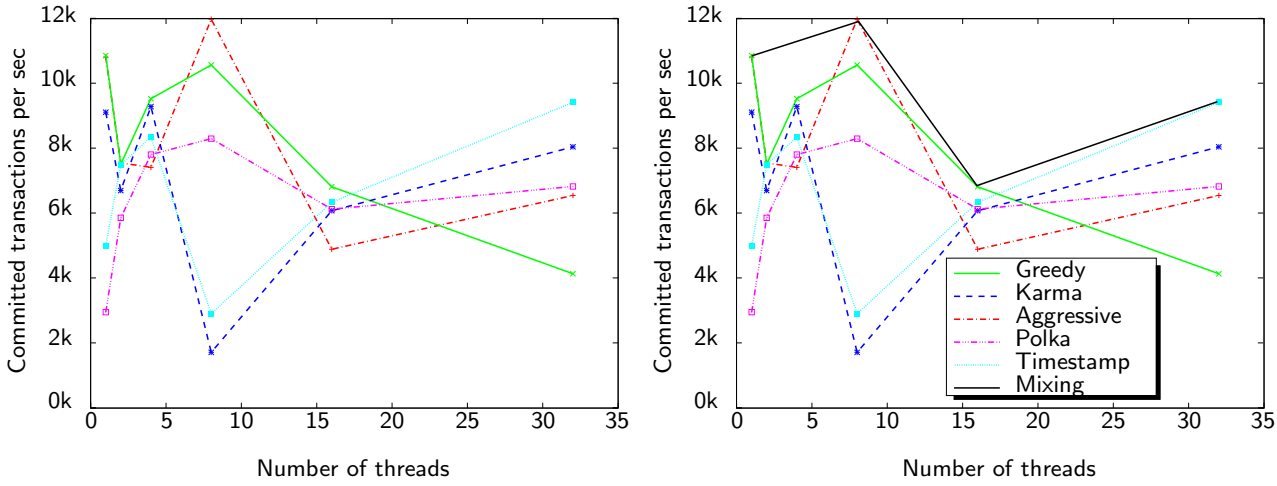
Figure 4 illustrates, following the example of Section 2, how we create a transaction from the Increment method, and associate with this transaction a Greedy contention manager [5].

3.3 Benchmarks

The benchmarks in this section provide some guidelines for choosing adequate contention managers in different parts of a given concurrent application. In fact, a programmer is encouraged to experiment with different contention management policies, especially since safety is not impacted.

Figures 1 and 5 show the number of committed transactions in a constant period of one second with respect to the number of threads, ranging from 1 to 32, with different contention management policies and in three different scenarios. Figure 1 depicts, on the left, a red-black tree application with low contention among transactions and, on the right, a red-black tree application with high contention among transactions. Figure 5 depicts a red-black forest application, a data structure made of fifty red-black trees, in which threads continuously insert and remove elements, in either one or all trees on a random basis; the length distribution of the transactions produced which a red-black forest exhibits a high variance.

As conveyed by the left part of Figure 1, when there is no contention among transactions at the end of the computation, for various length of uncontented periods, and transactions are approximately of the same size, the Greedy contention manager [5] provides the best throughput. Intuitively, this is because Greedy does not maintain costly data structures for assigning priority to transactions. On the other hand, the right part of Figure 1 shows that Karma [16] and Polka [17] provide a better throughput in a contention-intensive scenario. This might be explained by the fact that the priority assigned to transactions, though more costly to update, reveals itself a good estimator of the intuition that a transaction that has performed a lot of work should have a higher priority than a transaction that has performed less work.



(a) Red-black forest, high contention

(b) Adapting the contention manager to the number of threads

Fig. 5. Red-black forest application

In Figure 5(a), transactions are of irregular length, and the best contention management policies seem uncertain. More precisely, it highly varies depending on the number of threads (we come back to this in Section 4).

4 Polymorphic Contention Management

4.1 Mixing Contention Managers

In SXM, each transaction may be associated with a distinct contention manager class. Obviously, we would like to associate with each transaction within an application the contention manager for which the best throughput is obtained. This is not necessarily the same contention manager class for all (groups of) transactions.

Figure 5(b) shows the throughput of committed transactions per seconds when the contention manager associated with any new transaction adapts to the current number of threads, within the same application.

To illustrate this situation, consider a server responding to client requests over the Internet, designed such that each client is served by a different thread within the server. Consider that at some point, a high number of clients simultaneously send a request to the server. The number of threads within the application is then high, and the application benefits from switching the default contention manager to be used for any new transaction, to one that is efficient with a high number of threads. When the number of clients later decreases, the number of threads on the server decreases. The application then benefits from switching the default contention manager to one that is efficient with a low number of threads.

A programmer may also want to implement her own contention managers, for the purpose of her application. On the other hand, she might also want to use a contention manager that already exists for other transactions.

Clearly, addressing the conflict resolution (a priori) by considering every possible pair of contention managers in the ResolveConflict method is simply not possible. In the following, we discuss how that can be done in a dynamic manner.

4.2 Coping with Diversity

When two or more contention manager classes are mixed within a single application, two conflicting transactions are not necessarily associated with the same contention manager class. Consider for example a transaction T_a managed by a Greedy contention manager denoted M_a and a transaction T_b, attacking T_a, managed by a Karma contention manager denoted M_b. In the ResolveConflict method, M_b needs to

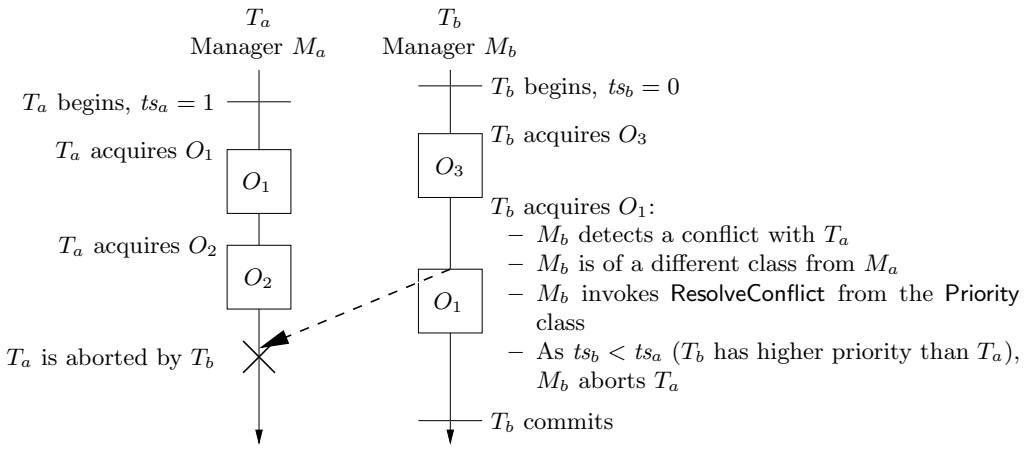


Fig. 6. T_b is attacking victim transaction T_a with a contention manager of a different class

decide whether to abort the victim transaction T_a, monitored by M_a. Although M_b has a reference to M_a, M_b does not know the dynamic type of M_a (Greedy). In fact, M_a is available to M_b as a reference of type IContentionManager, a simple interface implemented by all contention manager implementations in SXM.

To cope with the diversity of contention managers, the easiest policy we may imagine is that any contention manager immediately aborts a conflicting transaction managed by a different contention manager. In fact, this is the behavior of the Aggressive contention manager [16]. In this case, any contention manager is an Aggressive contention manager in face of a contention manager of an unknown class.

We classify contention managers in a hierarchy. The hierarchy is based on the cost associated to each contention manager, from the less costly ones (those that do not use any bookkeeping) to the most costly ones (those that maintain much information about the current transaction, the other transactions, etc.). The hierarchy is shown in Figure 7, and includes the contention managers defined in [5] and [16].

More precisely, Figure 7 depicts three general groups of contention managers: (a) ad-hoc contention managers which always adopt the same strategy independently of the computation of transactions, (b) contention managers which base their decision on information local to transactions, and (c) contention managers which base their decision on the computation and previous interactions of transactions. This is reflected in the implementation of the contention managers by (a) no variable at all, (b) simple data structures, and (c) complex data structures. Because contention managers in group b maintain simpler data structures and less bookkeeping than contention managers in group c, but, at the same time, base their decisions on elements of the computation of the transaction, in contrast with contention managers in group a, we use this class as the common denominator among all contention managers.

Furthermore, two transactions that are associated with distinct contention managers were probably not planned to conflict initially. Hence using contention management policies such as Karma or Polka to address the conflict does not really make sense in this case, because the number of transactional objects that have been accessed so far is probably not comparable. Making a decision based on the number of objects acquired by each transaction does not really reflect the priority among transactions.

To cope with this issue, we defined an abstract Priority contention manager, from which every concrete contention manager class inherits. This contention manager class associates a priority to every transaction, when the transaction is run. The Priority contention manager class exports a concrete conflict resolution ResolveConflict method, based on this priority.

Consider the scenario depicted in Fig. 6, where a transaction T_b attacks a transaction T_a, and denote by M_a (resp. M_b) the contention manager of T_a (resp. T_b), and by ResolveConflict$_a$ (resp. ResolveConflict$_b$) the original ResolveConflict of contention manager M_a (resp. M_b). When the SXM library detects the conflict, it invokes the ResolveConflict on M_b. Within the method, the conflict resolution algorithm now works as follows:

1. If M_b and M_a are of the same class, or M_b is of a superclass of M_a, then apply ResolveConflict$_b$ between T_a and T_b.

Rank		Contention manager	Data structures
0	a	IContentionManager	—
1		Aggressive, Polite	—
2	b	Greedy, Killblocked	Birthdate
3		(Published)Timestamp	Birthdate, variable
4	c	Kindergarten	List of transactions
5		Karma, Polka, Eruption	List of objects

Fig. 7. A hierarchy of contention managers

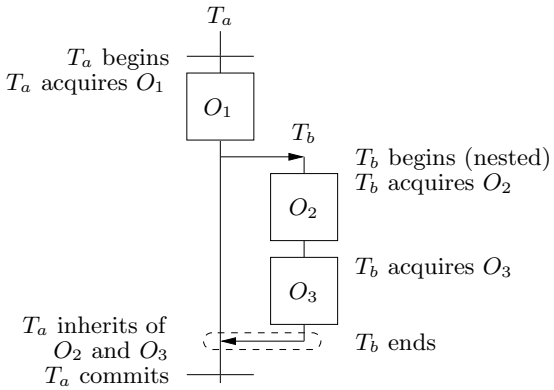


Fig. 8. Nesting transactions

2. If M_b and M_a are of incomparable classes, then apply the ResolveConflict method as defined by the Priority contention manager, between T_a and T_b.

We use the priority and conflict resolution algorithm of the Greedy contention manager as the common priority and conflict resolution algorithm of the Priority contention manager. The Greedy contention manager records the time at which a transaction starts for the first time. When two transactions conflicts, these timestamps are used as priority, with the idea that older transactions have higher priority than recent transactions. Consider a victim transaction T_a and a transaction T_b attacking T_a. We associate timestamp ts_a (resp. ts_b) with T_a (resp. T_b). The ResolveConflict method of the Greedy contention manager proceeds as follows (recall that T_b is attacking T_a):

1. If $ts_a > ts_b$ (T_a was started more recently than T_b) or T_a is waiting, then abort T_a.
2. Otherwise, wait until T_a commits, aborts or starts waiting. (If T_a starts waiting, then see Rule 1.)

5 Associating Contention Managers with Transactions

In Section 3 we pointed out how a particular contention manager may be associated with a transaction at the top level. We go one step further, and allow for decomposing a transaction into nested transactions (following the method invocation tree), and associating different contention managers within the same top level transaction. We discuss now the issues behind nesting transactions and explain how each nested transaction may be associated with a distinct contention manager.

5.1 Issues behind Nesting Transactions

In SXM, the transaction boundaries are mapped on the method boundaries. Consider an application where a method A invokes a method B. In the application, we declare transaction T_a (resp. T_b) associated with method A (resp. B). There are two possibilities for running transaction T_b:

1. T_b runs within transaction T_a.
2. T_b runs as a separate transaction nested in T_a.

Running transaction T_b as a nested transaction (possibility 2) is more costly than running T_b within T_a (possibility 1), since it requires creating a fresh transaction state. (We give more details on what it takes to create a transaction in Section 6.) To illustrate the usefulness of possibility 2, assume that before invoking T_b, T_a performs a long computation consuming a lot of resources. If a conflict is encountered when executing T_b with a third transaction T_c, we would prefer restarting T_b without restarting T_a. In this case, it is convenient to run T_b as a separate transaction, which may be aborted and restarted separately from T_a. In this case, T_a is not impacted by the conflict encountered by T_b, and the execution of T_a may resume when T_b eventually commits.

Whether one approach is more appropriate than the other depends on the context. Hence we consider both approaches, as either one may be appropriate in different situations: our SXM library provides the

```
1: static public void Main(string argv[ ])
2:     Application application = new Application();
3:     bool runAsNestedTransaction = true;
4:     Delegate delegate = new SXMDelegate(application.Increment);
5:     SXMAction incrementAction = SXMAction.Create(delegate,typeof(Greedy),runAsNestedTransaction);
6:     ...
7:     incrementAction.Run();
```

Fig. 9. Specifying nested transaction semantics

programmer with the possibility, for every transaction, to choose which approach to use. When creating a transaction from a method, the programmer may specify, using a boolean parameter, whether the transaction should run within its parent transaction, or as a nested transaction. Figure 9 shows the syntax for declaring a transaction which should be nested in a parent transaction.

In terms of contention management, SXM defines on the one hand notification methods specific to the case where a transaction is run within another transaction, for instance to inform the contention manager about the depth of the transaction. On the other hand, when running a transaction T_b nested in a transaction T_a, SXM enables to associate with T_b a contention manager that is distinct from the contention manager of T_a.

Whereas the implementation of possibility 1 is trivial and only requires declaring additional notification methods, the implementation of possibility 2 is more involved. With possibility 2, if T_b aborts, our SXM library restarts it, without impacting T_a, whereas if T_b reaches completion without aborting, then T_b returns the thread of execution to T_a. When T_b terminates, T_b cannot really commit, because its parent transaction T_a is not finished yet, and may still be aborted later on. If T_b commits, and T_a is later aborted, we would have to roll back the changes of T_b, which is cumbersome.

In SXM, when any nested transaction finishes, the parent transaction inherits the objects acquired by the nested transaction, as shown in Figure 8. When the parent transaction later commits, all the objects it has acquired, included those inherited from nested transactions, are committed. (We give more details on the actual implementation of nested transactions in Section 6.)

5.2 Interactive Contention Management

As part of resolving a conflict among two conflicting transactions, a contention manager may perform several actions: abort the victim transaction, backoff for a random or exponential time, access data structures for bookkeeping, change the priority of the attacking or of the victim transaction, etc.

SXM features the Interactive contention manager, that returns the thread of execution to the parent transaction (or to the executing thread if already at the top level), as soon as a nested transaction aborts. This allows for defining more complex schemes, for instance trying an alternative transaction in case of the first transaction aborts (similarly to the orElse construct of [7]). This is key to dynamically changing the contention manager of the nested transaction, if one feels that another contention manager would then perform better.

The Interactive contention manager is parametrized with another contention manager, as the Interactive contention manager does not feature any contention management policy on its own. Figure 10 depicts an example of the Interactive contention manager. In this example, a parent transaction is associated with the Interactive contention manager (line 9), which was previously parametrized with the Aggressive contention manager (line 5), and a nested transaction is associated with the Greedy contention manager (line 10). The nested transaction is created in such a way it will be run as a separate transaction by SXM (line 10). In case the nested transaction aborts, the Interactive contention manager throws an exception. In the exception handler (line 8, second column), the parent transaction assigns another contention manager class to the nested transaction (line 12, second column), before restarting it (while loop at line 4, second column). When the nested transaction succeeds, the parent transaction may resume its execution.

6 Implementation of SXM in C#

In SXM, when a transaction invokes an operation on a transactional object, a synchronization code is transparently executed before the operation is effectively performed. This synchronization code is generated and added to the program at runtime, using the reflexive API of C#. Roughly speaking, this

```
1: class Nesting
2:    static SXMAction parentAction;
3:    static SXMAction childAction;

4:    static public void Main(string[ ] argv)
5:       Interactive.ManagerType = typeof(Aggressive);
6:       Nesting example = new Nesting();
7:       Delegate parent = new
                 SXMDelegate(example.ParentMethod);
8:       Delegate child = new
                 SXMDelegate(example.ChildMethod);
9:       parentAction = SXMAction.Create(parent,
                 typeof(Interactive));
10:      childAction = SXMAction.Create(child,
                 typeof(Greedy),true);
11:      ...
12:      parentAction.Run();
```

```
1:  public void ParentMethod()
2:    ...
3:    // Repeat while nested transaction aborts
4:    while true do
5:       try
6:          childAction.Run();
7:          break;
8:       catch InteractiveException
9:          ...
10:         // Associate another manager
11:         // with child transaction if it aborts
12:         childAction.ManagerType = typeof(Karma);
13:         ...

14:  public void ChildMethod()
15:    ...
```

Fig. 10. The Interactive contention manager

code allows for detecting conflicts among transactions, upon acquiring a transactional object. In this section, we give more details on implementation issues in SXM.

6.1 Transactional Object Structure

For an object to be transactional, its class is marked with the Atomic attribute and implements the ICloneable interface. Get properties within this class are (implicitly) considered as *read* operations, whereas set properties are (implicitly) considered as *write* operations. The fields of an Atomic class are declared private, so that they are not accessible from the outside of the object by accident, preventing the transaction abstraction from being broken.

To create a transactional object in a program, maybe in a transaction, a thread uses a *transactional object factory*, SXMObjectFactory. Behind the scenes, a transactional object of class Type is represented by an instance of class TX_Type, which inherits from Type. Class TX_Type is created at runtime by the SXMObjectFactory, which is given class Type as a parameter, and return class TX_Type. As TX_Type inherits from Type, instances of class TX_Type returned by the factory may be referenced as instances of class Type, hence a full transparency for the programmer.

An instance of class TX_Type has a single field, denoted stmObject, of type SynchState.[4] Roughly speaking, the instance of class SynchState supports the methods necessary for acquiring a copy of the object. The SynchState object references a Locator object. A Locator references in its turn an XState instance and two instances of class Type. An XState object represents the state of the transaction, and may have three values: ACTIVE, COMMITTED or ABORTED. When a fresh XState instance is created for a transaction, it is in the ACTIVE state. In a Locator instance, the XState instance corresponds to the state of the transaction which acquired the transactional object the most recently.

Initially, the SynchState instance referenced by TX_Type, references a Locator instance containing the COMMITTED transaction state, and the original version of the transactional object is referenced by the new object. Figure 11 illustrates the structure of a transactional object.

When a transaction attempts to invoke a method on the transactional object, the transaction really invokes the method with the same signature on the TX_Type instance. (Recall that each instance of a class marked with the Atomic attribute is created from the SXMObjectFactory, which returns instances of class TX_Type.)

TX_Type declares the same properties, with the same signature, as Type. The body of a get (resp. set) property in class TX_Type adds synchronization code before calling the original get (resp. set) property. More precisely, it is implemented as follows:

1. An invocation of OpenRead (resp. OpenWrite) method on stmObject object is performed. Roughly speaking, this encapsulates the steps necessary to obtain a fresh copy of the object, and to make sure

[4] The SynchState object corresponds to the TMObject in the DSTM system of [10].

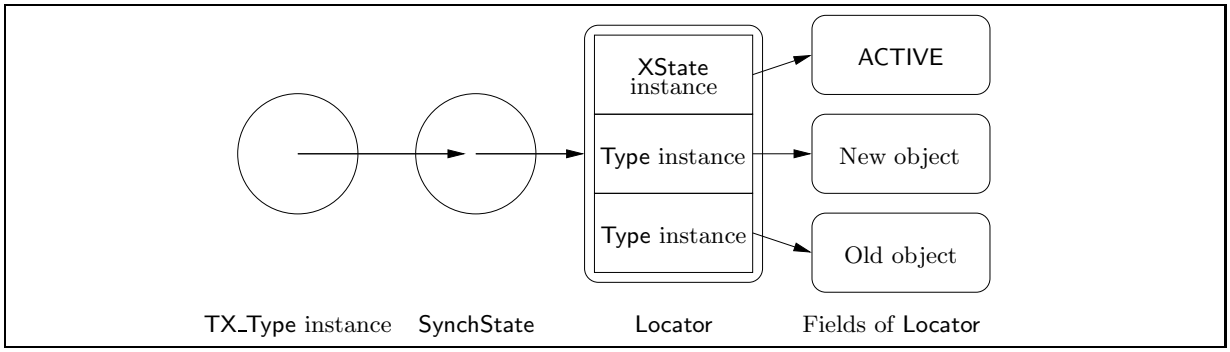


Fig. 11. Transactional object structure

```
1: [Atomic]
2: class Node : ICloneable
3:    private Node next;
4:    private int element;

5:    public Node(Node next, int element)
6:      this.next = next;
7:      this.element = element;

8:    public object Clone()
9:      return new Node(this.next,this.element);

10:   property Node Next
11:     get
12:       return this.next;
13:     set
14:       this.next = value;

15:   property int Element
16:     get
17:       return this.element;
18:     set
19:       this.element = value;
```

(a) Class Node

```
1: class TX_Node : Node
2:    private SynchState stmObject;

3:    public TX_Node() : base()
4:      this.stmObject = new SynchState(this);

5:   property Node Next
6:     get
7:       Node target = stmObject.OpenRead();
8:       return target.Next;
9:     set
10:      Node target = stmObject.OpenWrite();
11:      target.Next = value;

12:  property int Element
13:    get
14:      Node target = stmObject.OpenRead();
15:      return target.Element;
16:    set
17:      Node target = stmObject.OpenWrite();
18:      target.Element = value;
```

(b) Class TX_Node, generated from Node at runtime

Fig. 12. Classes Node and TX_Node

no other transaction has a copy of the object. Corresponding to the contention management policy, this means to either abort a conflicting transaction, or to send its transaction to sleep for some time.

2. The original get (resp. set) property of class Type is invoked on the copy returned, and the result is returned to the user (resp. the copy is modified with the given value).

For instance, consider a linked list data structure. A list is composed of zero or more nodes, represented by class Node. A node contains two fields, a integer element, representing the value stored in the node, and a reference next, on the next node in this list. Both fields are accessed through get (resp. set) properties, which only return the value in the field (resp. set the field with the new value). Node is shown in Figure 12(a) and TX_Node generated from Node at runtime through the SXMObjectFactory is shown in Figure 12(b).

We discuss the OpenWrite method, the OpenRead method works in an analogous way.

In OpenWrite, a reference to the Locator installed in the SynchState instance is obtained first. In the Locator, we may read the status (i.e., the XState field) of the transaction that most recently updated the object, to determine which of the old or new object is the current version of the object: if the status

of the transaction is COMMITTED, then the new version is the current version, whereas if the status is ACTIVE or ABORTED, then the old object is the current version.

We instantiate a fresh Locator object, with an ACTIVE status field, and a clone of the current version of the object (as determined above) referenced the *old* object (recall that the object supports the ICloneable interface).

If the state of the last transaction that updated the object is ACTIVE, we are in presence of a conflict. In this case, the transaction calls ResolveConflict on its contention manager. If the contention manager decides to abort the other transaction, it atomically swaps the status of the other transaction, from ACTIVE to ABORTED. The contention manager may otherwise decides to wait for some time, and retry from the beginning of the OpenWrite method.

After aborting any conflicting transaction, the transaction atomically updates the SynchState instance (with compare-and-swap) to install the newly created Locator in place of the previous Locator.

At last, the transaction commits by atomically swapping its state from ACTIVE to COMMITTED. If this succeeds, all the new versions of the objects accessed by the transaction become the current versions. Committing fails if another transaction has already swapped the state to ABORTED.

The OpenRead method implementation differs from the OpenWrite method implementation, in that two transactions *reading* the same transactional objects do not conflict with each other.

Note that the indirection induced by the SynchState instance is not strictly necessary. In fact, we could program the full OpenRead and OpenWrite methods directly within TX_Type. However, this would require to write the full body of OpenRead and OpenWrite at runtime by using the C# reflection API. For the sake of simplicity, we introduce this extra object indirection.

6.2 Transaction Structure

Upon running a transaction, by invoking the Run method of the SXMAction object, SXM creates a new transaction state (a fresh XState instance in the ACTIVE state), and associates this new transaction state instance with the new transaction. The transaction state is then later available, when the transaction later acquires transactional object (through OpenRead and OpenWrite methods).

When a transaction starts, a depth counter is incremented. A *zero* value corresponds to the transaction at the top level. If the depth is more than *zero*, then the parameter provided by the programmer is used to determined whether to run the transaction as a nested transaction and associate with it a new contention manager instance, or to run the transaction within the parent transaction. SXM then invokes the transactional method by simply invoking the delegate. When the delegate returns, SXM tries to atomically commit the modifications (by swapping the state of the transaction from ACTIVE to COMMITTED). If the transaction fails to commit, then SXM creates a fresh XState instance, associates this instance with the transaction, and invokes the delegate once again.

If a nested transaction aborts, this means that its state object has been swapped from ACTIVE to ABORTED. However, the parent transaction, which does not share its state with nested transactions, is not affected. Hence SXM only restarts the nested transaction.

If the nested transaction ends without aborting, the parent transaction inherits the objects accessed by the nested transaction. To achieve this, SXM modifies, one after another, the Locator objects referencing the transactional objects accessed by the nested transaction. The modification consists in changing the transaction state reference, from the nested transaction state, to the parent transaction state (for which we keep a reference within the nested transaction).

- If SXM notices that, after having modified the Locator of every transactional object accessed by the nested transaction, the transaction state of the nested transaction has status ABORTED, then SXM aborts the parent transaction (it atomically swaps the state of the parent transaction from ACTIVE to ABORTED), and then restarts the parent transaction from the beginning.
- If SXM succeeds in modifying each Locator object of the transactional objects acquired by the nested transaction, without the nested transaction being aborted, then SXM returns the thread of execution to the parent transaction, which may continue executing. In this case, every transactional object modified by the nested transaction now references the parent transaction in its Locator object.

When the parent transaction ends, SXM tries to commit the transaction by atomically swapping its state from ACTIVE to COMMITTED. This signals, for each transactional object modified by the transaction and now including the objects inherited from nested transactions, that the new object is the

```
1: class Application
2:    Counter counter;
3:    public Application()
4:       this.counter = new Counter();
5:    [Transactional(Greedy,true)]
6:    public void Increment()
7:       this.counter.increment;
8:    static public Main(string[ ] argv)
9:       Application application = new Application();
10:      application.Increment();
```

```
1: [Atomic]
2: class Counter
3:    private int balance;
4:    property int increment
5:       set
6:          balance = balance + 1;
```

Fig. 13. A postprocessing approach to an example of a counter in SXM

current version of the transactional object. If SXM fails to commit (indicating contention with another transaction), SXM restarts the parent transaction from the beginning.

6.3 Postprocessor Approach to Declaring Transactions

SXM could also be ported as an extension of the CIL postprocessor. CIL is the common intermediate language emitted by the C# compiler. More precisely, we propose extensions to the postprocessor for declaring transactional methods in SXM.

The Transactional attribute marks methods that are transactional. The transaction starts when the method begins and ends when the method ends. As before, the Atomic class attribute marks the shared objects that may be accessed by transactional methods. (The ICloneable interface is implemented automatically with the Atomic attribute.)

Figure 13 revisits the example of the counter, implemented in SXM with postprocessing extensions. The class Application defines a transactional method Increment, which acts on an instance of the Counter class. The Counter class is declared with the Atomic attribute.

To specify which contention manager type to use with a transaction, the programmer may give a parameter to the Transactional attribute. The programmer may also specify, with a parameter to the Transactional attribute, whether a transaction should be executed as a nested transaction (true), or within the parent transaction (false), in case the transaction is run as a nested transaction.

7 Concluding Remarks

In [10], Herlihy et al. proposed a dynamic software transaction memory (DSTM) system in Java for transactions accessing a set of objects not fixed or known in advance. In DSTM, a single contention manager class is used to monitor all transactions.

Harris and Fraser described in [7] a transaction scheme resembling conditional critical regions (CCR). They also proposed a simple form of nesting transactions, where committing a transaction occurs only when the top-level transactions returns. The contention manager is fixed, and a transaction that encounters a conflict systematically aborts, after waiting for the conflicting transaction to finish.

Harris et al. proposed in [8] a STM system in Concurrent Haskell, in which transactions are declared using an atomic block. Transactions can be composed while preserving the atomicity of the composition. Contention management was not discussed.

Scherer and Scott compared in [16] many contention managers, considering various kind of metrics for prioritizing transactions. They elected one (Polka) as the best contention manager and did not consider the question of integrating different contention managers within a single application when the concurrency pattern varies over the life of the application. Taking into account alternative contention managers (Greedy [5]) as well as different load situations led us to revisit the "universality" of Polka, and introduce our polymorphic structure.

The way transactions are nested and mapped to method boundaries in SXM resembles that of Argus [12]. Argus introduced object wrappers called *guardians*. A guardian object is similar to a transactional object as defined in this paper, and encapsulates objects to be accessed within a transaction to provide atomicity guarantees. Argus uses a lock-based approach to ensure atomicity of transactions.

Our transaction scheme is more pragmatic than in Argus as it leaves it up to the programmer to decide whether the method invocation runs in a nested transaction or remains within the parent. Moreover, we only pay the price of nesting transaction upon usage. In this sense, our scheme is closer to that of ACS [4]. Contention management was however not factored out neither in Argus nor in ACS, and concurrency control was achieved using locking: when a nested transaction returns, the parent inherits the locks.

Modular concurrency control approaches [3, 18, 20] considered the semantics of the operations to enable transaction interleaving. High-level atomicity is preserved, independently of the order in which commutative atomic operations are executed. On the other hand, identifying an operation as commutative when it is not, may lead to a violation of safety; whereas safety is always guaranteed in a STM application. In a sense, a contention manager in a STM application extracts a part of concurrency control that is only concerned with progress (and cannot hamper safety).

References

1. E. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.
2. J. Gray. A transaction model, automata languages and programming. *Lecture Notes in Computer Science*, 85:282–298, 1980.
3. R. Guerraoui. Atomic object composition. In *ECOOP'94: Proceedings of the European Conference on Object-Oriented Programming*, pages 118–138. Springer-Verlag, 1994.
4. R. Guerraoui, R. Capobianchi, A. Lanusse, and P. Roux. Nesting actions through asynchronous message passing: the ACS protocol. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 170–184. Springer-Verlag, 1992.
5. R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of contention managers. In *PODC'05: Proceedings of the twenty-fourth annual symposium on Principles of Distributed Computing*. ACM Press, 2005.
6. L. Hammond, B. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.
7. T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA'03: Proceedings of the eighteenth ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2003.
8. T. Harris, S. Marlow, S. Jones, and M. Herlihy. Composable memory transaction. Technical report, Microsoft Research Cambridge, December 2004.
9. M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS '03: Proceedings of the twenty-third International Conference on Distributed Computing Systems*, page 522. IEEE Computer Society, 2003.
10. M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC'03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM Press, 2003.
11. M. Herlihy and J. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA'93: Proceedings of the twentieth Annual International Symposium on Computer Architecture*, pages 289–300. ACM Press, 1993.
12. B. Liskov. Distributed programming in argus. *Communication of ACM*, 31(3):300–312, 1988.
13. J. E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, MIT, 1981.
14. C. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.
15. Microsoft Research. *C# software transactional memory*. http://research.microsoft.com/research/downloads/default.aspx.
16. W. Scherer and M. Scott. Contention management in dynamic software transactional memory. In *Workshop on Concurrency and Synchronization in Java Programs*, July 2004.
17. W. Scherer and M. Scott. Advanced contention management for dynamic software transactional memory. In *PODC'05: Proceedings of the twenty-fourth annual symposium on Principles of Distributed Computing*. ACM Press, 2005.
18. P. Schwarz and A. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, 1984.
19. W. Weihl. *Specification and Implementation of Atomic Data Types*. PhD thesis, MIT, 1984.
20. W. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, 1989.