Memoizing Multi-Threaded Transactions

Lukasz Ziarek   Suresh Jagannathan

Department of Computer Science Purdue University
[lziarek,suresh]@purdue.edu

Abstract. There has been much recent interest in using transactions to simplify concurrent programming, improve scalability, and increase performance. When a transaction must abort due to a serializability violation, deadlock, or resource exhaustion, its effects are revoked, and the transaction re-executed. For long-lived transactions, however, the cost of aborts and subsequent re-executions can be prohibitive. To ensure performance, programmers are often forced to reason about transaction lifetimes and interactions while structuring their code, defeating the simplicity transactions purport to provide.

One way to reduce the overheads of re-executing a failed transaction is to avoid re-executing those operations that were unaffected by the violation(s) that induced the abort. Memoization is one way to capitalize on re-execution savings, especially if violations are not pervasive. Abstractly, if a procedure p is applied with argument v within a transaction, and the transaction aborts, p need only be re-evaluated when the transaction is retried if its argument is different from v.

In this paper, we consider the memoization problem for transactions in the context of Concurrent ML (CML) [20]. Our design supports multi-threaded transactions which allow internal communication through synchronous channel-based communication. The challenge to memoization in the context is ensuring that communication actions performed by memoized procedures in the original (aborted) execution can be satisfied when the transaction is retried.

We validate the effectiveness of our approach using STMBench7 [9], a customizable transaction benchmark. Our results indicate that memoization for CML-based transactions can lead to substantial reduction in re-execution costs (up to 45% on some configurations), with low memory overheads.

1 Introduction

Concurrency control mechanisms, such as transactions, rely on efficient control and state restoration mechanisms for performance. When a transaction aborts, due to a serializability violation [8], deadlock, transient fault [24], or resource exhaustion [11], its effects are typically undone, and the transaction retried. A long-lived transaction that aborts represents wasted work, both in terms of the operations it has performed whose effects must now be erased, and in terms of overheads incurred to implement the concurrency control protocol; these overheads include logging costs, read and write barriers, contention management, etc. [12].

Transactional abstractions embedded in functional languages (e.g., AtomCaml [21], proposals in Scheme48 [14], or STM Haskell [11]) benefit from having relatively few stateful operations, but when a transaction is aborted, the cost of re-execution still remains. One way to reduce this overhead is to avoid re-executing those operations that

would yield the same results produced in the original failed execution. Consider a transaction T that performs a set of operations, and subsequently aborts. When T is re-executed, many of the operations it originally performed may yield the same result, because they were unaffected by any intervening global state change between the original (failed) and subsequent (retry) execution. Avoiding re-execution of these operations reduces the overhead of failure, and thus allows the programmer more flexibility and leeway to identify regions that would benefit from being executed transactionally.

Static techniques for eliminating redundant code, such as subexpression elimination or partial redundancy elimination, are ineffective here because global runtime conditions dictate whether or not an operation is redundant. Memoization [15, 18] is a well-known dynamic technique used to eliminate calls to pure functions. If a function f supplied with argument v yields result v', then a subsequent call to f with v can be simply reduced to v' without re-executing f's body, *provided* that f is effect-free.

In this paper, we consider the design and implementation of a memoization scheme for an extension of Concurrent ML [20] (CML) that supports multi-threaded transactions. CML is particularly well-suited for our study because it serves as a natural substrate upon which to implement a variety of different transactional abstractions [7]. In our design, threads executing within a transaction communicate through CML synchronous events. Isolation and atomicity among transactions are still preserved. Multi-threaded transactions can, thus, be viewed as a computation which executes atomically and in isolation instead of a simple code block. Our goal is to utilize memoization techniques to avoid re-execution overheads of long-lived multi-threaded transactions that may be aborted.

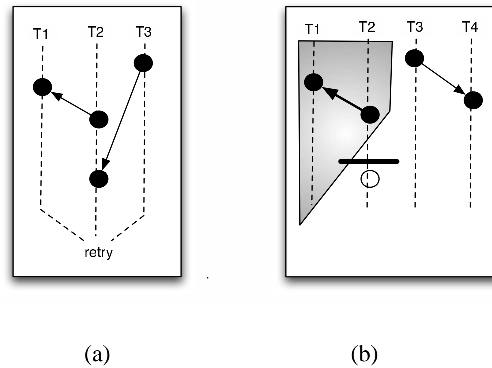
The paper is organized as follows. In the next section, we describe our programming model, and introduce issues associated with memoization of synchronous communication actions. Section 3 provides additional motivation. We introduce *partial memoziation*, a refinement that has substantial practical benefits in Section 4. Implementation details are given in Section 5. We present a case study using STM-Bench [9], a highly-concurrennt transactional benchmark, in Section 6. Related work and conclusions are given in Section 7.

2 Programming Model

Our programming model supports multi-threaded closed nested transactions. An expression wrapped within an `atomic` expression is executed transactionally. If the value yielded by a transaction is `retry`, the transaction is automatically re-executed. A transaction may `retry` because a serializability violation was detected when it attempted to commit, or because it attempts to acquire an unavailable resource [11]. An executing transaction may create threads which in turn may communicate with other threads executing within the transaction using synchronous message passing expressed via CML selective communication abstractions. To enforce isolation, communication between threads executing within different transactions is not permitted. A transaction attempts to commit only when all threads it has spawned complete. Updates to shared channels performed by a transaction are not visible to other transactions until the entire transaction completes successfully.

We are interested in allowing multi-threaded transactions primarily for reasons of composability and performance. A computation wrapped within an atomic section may invoke other procedures that may spawn threads and have these threads communicate with one another. This is especially possible when considering long-lived transactions that encapsulate complex computations involving multiple layers of abstraction. Prohibiting such activity within an atomic section would necessarily compromise composability. Moreover, allowing multi-threaded computation may improve overall transaction performance; this is certainly the case in the benchmark study we present in Section 6. Inter-thread communication within a transaction is handled through dynamically created channels on which threads place and consume values. Since communication is synchronous, a thread wishing to communicate on a channel that has no ready recipient must block until one exists. Communication on channels is ordered.

2.1 Memoization



(a) (b)

Fig. 1. Threads are represented as dotted lines while circles define communication points. The shaded area depicts computation which can be memoized based on communications which are satisfiable.

A transaction may spawn a collection of threads that communicate with one another via message-passing (see Fig. 1(a)). When the transaction is retried (see Fig. 1(b)), some of these communication events may be satisfiable when the procedures in which the events occurred are invoked (e.g., the first communication event in the first two threads), while others are not (e.g., the second communication action performed by thread T2 to thread T3). The shaded region indicates the pure computation in T1 and T2 that may be avoided when the transaction is re-executed. Note that T2 must resume execution from the second communication action because the synchronization action with T3 from the aborted execution is not satisfiable when the transaction is retried; since the send action by T3 was received by T4, there is no other sender available to provide the same value to T2.

In this context, deciding whether an application of procedure f can be avoided based on previously recorded memo information depends upon the value of its arguments, the

communication actions performed by f, threads f spawns, and f's return value. Thus, the memoized return value of a call to f can be used if (a) the argument given matches the argument previously supplied; (b) recipients for values sent by f on channels in an earlier call are still available on those channels; (c) a value that was consumed by f on some channel in an earlier call is again ready to be sent by another thread; and (d) threads created by f can be spawned with the same arguments supplied in the memoized version. Ordering constraints on all sends and receives performed by the procedure must also be enforced.

To avoid performing the pure computation within a call, a send action performed within the applied procedure, for example, will need to be paired with a receive operation executed by some other thread. Unfortunately, there may be no thread currently scheduled that is waiting to receive on this channel. Consider an application that calls a memoized procedure f which (a) creates a thread T that receives a value on channel c, and (b) sends a value on c computed through values received on other channels that is then consumed by T. To safely use the memoized return value for f nonetheless still requires that T be instantiated, and that communication events executed in the first call can still be satisfied (e.g., the values f previously read on other channels are still available on those channels). Ensuring these actions can succeed involves a systematic exploration of the execution state space to induce a schedule that allows us to consider the call in the context of a global state in which these conditions are satisfied.

Because such an exploration may be infeasible in practice, our formulation considers a weaker alternative called *partial* memoization. Rather than requiring global execution to reach a state in which *all* constraints in a memoized application are satisfied, partial memoization gives implementations the freedom to discharge some fraction of these constraints, performing the rest of the application as normal.

3 Tracking Communication Actions

The key requirement for effective memoization of procedures executing within CML transactions is the ability to track communication actions performed among procedures. Provided that the global state would permit these same actions to succeed if a procedure is re-executed with the same inputs, memoization can be employed to reduce re-execution costs.

```
atomic(fn () =>
       let val  (c1, c2) = (mkCh(), mkCh())
           fun  f() = (...; send(c1, v1); ...)
           fun  g() = (recv(c1); send(c2,v2))
       in spawn(f()); spawn(g()); recv(c2)
       end)
```

Fig. 2. The call to `f` can always be memoized since there is only a single receiver on channel `c1`.

Consider the example code presented in Fig. 2 that spawns two threads to execute procedures `f` and `g` within an atomic section. Suppose that the section fails to commit,

and must be retried. To correctly utilize f 's memoized version from the original failed execution, we must be able to guarantee the value sent on channel c1 has a recipient. At the time the memoization check is performed, the thread computing g may not even have been scheduled. However, by delaying the memoization decision for f 's call until g is ready to receive a value on c1 , we guarantee that memoized information stored for f can be successfully used to avoid performing the pure computation within its body.

```
atomic(fn () =>
        let val (c1, c2, c3) =
                (mkCh(), mkCh(), mkCh())
            fun f() = (...; send(c1,v1); recv(c2))
            fun g() = (recv(c1); recv(c2))
            fun h() = (send(c2,v2);
                       send(c2,v3);
                       send(c3,()))
        in (spawn(f()); spawn(g()); spawn(h());
            recv(c3))
        end)
```

Fig. 3. Because there may be multiple possible interleavings that pair synchronous communication actions among concurrently executing threads, memoization requires dynamically tracking these events.

Unfortunately, reasoning about whether an application can leverage memoized information is usually more difficult. Consider a slightly modified version of the program shown in Fig. 3, that introduces an auxiliary procedure h . Procedure f communicates with g via channel c1 . It also either receives value v2 or v3 from h depending upon its interleaving with g . Suppose that when this section is first executed, g receives values v2 from h and f receives value v3 . If the section must be re-executed, the call to f can be avoided only if the interleaving of actions between g and h allow f to receive v3 . Thus, a decision about whether the call to f can be elided requires also reasoning about the interactions between h and g , and may involve enforcing a specific schedule to ensure synchronous operations mirror their behavior under the aborted execution.

Notice that if v2 and v3 are equal, the receive in f can be paired with either send in h . Thus, memoization can be leveraged even under different schedules than a prior execution. Unlike program replay mechanisms [23], no qualifications are made on the state of the thread with which a memoization candidate communicates. Consequently, an application can utilize a memoized version of a procedure under a completely different interleaving of threads and need not communicate with the same threads or operations it did during its previous execution.

4 Approach

To support memoization, we must record, in addition to argument and return values, synchronous communication actions, thread spawns, channel creation etc. as part of

the memoized state. These actions define a set of constraints that must be satisfied at subsequent applications of a memoized procedure. To record constraints, we require expressions to manipulate a *memo store*, a map that given a procedure identifier and an argument value, returns the set of effects performed by the procedure when invoked with that argument. If the set of constraints returned by the memo store is satisfied in the current state, then the return value can be used, and the application elided. For example, if there is a communication constraint that expects the procedure to receive value `v` on channel `c`, and at the point of call, there exists a thread able to send `v` on `c`, evaluation can proceed to a state in which the sender's action is discharged, and the receive constraint is considered satisfied.

If the current constraint expects to send a value `v` on channel `l`, and there exists a thread waiting on `l`, the constraint is also satisfied. A send operation can match with any waiting receive action on that channel. The semantics of synchronous communication allows us the freedom to consider pairings of sends with receives other than the one it communicated with in the original memoized execution. This is because a receive action places no restriction on either the value it reads, or the specific sender that provides that the value. Similarly, if the current constraint records the fact that the previous application of the function spawned a new thread, or channel, then those actions must be performed as well. Thus, if all recorded constraints, which represent effects performed within a procedure *p*, can be satisfied in the order in which they occur, pure computation within the *p*'s body can be elided at its calls.

4.1 Partial Memoization

Determining whether all memoization constraints can be satisfied may require performing a potentially unbounded number of evaluation steps to yield an appropriate global state. However, even if it is not readily possible to determine if all constraints necessary to elide the pure computation within an application can be satisfied, it may be possible to determine that some prefix of the constraint sequence can be discharged. Partial memoization allows us to avoid re-executing any pure computation bracketed by the first and last elements of this prefix.

Consider the example presented in Fig 4. Within the atomic section, we apply procedures `f`, `g`, `h` and `i`. The calls to `g`, `h`, and `i` are evaluated within separate threads of control, while the application of `f` takes place in the original thread. These different threads communicate with one other over shared channels `c1` and `c2`.

Suppose the atomic section aborts, and must be re-executed. We can now consider whether the call to `f` can be elided when the section is re-executed. In the initial execution of the atomic section, spawn constraints would have been added for the threads responsible for executing `g`, `h`, and `i`. Second, a send constraint followed by a receive constraint, modeling the exchange of values `v1` and either `v2` or `v3` on channels `c1` and `c2` would have been included in the memo store for `f`. For the sake of the discussion, assume that the send of `v2` by `h` was consumed by `g` and the send of `v3` was paired with the receive in `f`.

The spawn constraints for the different threads are always satisfiable, and when discharged, will result in the creation of new threads which will begin their execution by trying to apply `g`, `h` and `i`, consulting their memoized versions to determine if all

```
atomic(fn () =>
   let val (c1,c2) = (mkCh(),mkCh())
       fun f ()  = (send(c1,v1); ... recv(c2))
       fun g ()  = (recv(c1); ... recv(c2))
       fun h () =  (...
                    send(c2,v2);
                    send(c2,v3));
       fun i () =  recv(c2)
   in spawn(g); spawn(h); spawn(i);
      f();  send(c2, v3)
      ...
      retry
   end)
end
```

Fig. 4. Determining if an application can be memoized may require examining an arbitrary number of possible thread interleavings.

necessary constraints can be satisfied. The send constraint associated with `f` matches the corresponding receive constraint associated found in the memo store for `g`. Determining whether the receive constraint associated with `f` can be matched requires more work. To match constraints properly, we need to force a schedule that causes `g` to receive the first send by `h` and `f` to receive the second, causing `i` to block until `f` completes.

Fixing such a schedule is tantamount to examining an unbounded set of interleavings. Instead, we could *partially* elide the execution of `f`'s call on re-execution by satisfying the send constraint (that communicates `v1` on `c1` to `g`), avoiding the pure computation following (abstracted by "..."), allowing the application of `f` to begin execution at the `recv` on `c2`. Resumption at this point may lead to the communication of `v2` from `h` rather than `v3`; this is certainly a valid outcome, but different from the original execution.

5 Implementation

Our implementation is incorporated within MLton [16], a whole-program optimizing compiler for Standard ML. The main changes to the underlying compiler and library infrastructure are the insertion of write barriers to track channel updates, barriers to monitor procedure arguments and return values, hooks to the CML library to monitor channel based communication, and changes to the Concurrent ML scheduler. The entire implementation is roughly 5K lines of SML: 3K for the STM, and 300 lines of changes to CML.

5.1 STM Implementation

Our STM implementation implements an eager versioning, lazy conflict detection protocol [4, 22]. References are implemented as "servers" operating across a set of channels; each channel has one server receiving from it and any number of channels sending to it. Our implementation uses both exclusive and shared locks to optimize read-only

transactions. If a transaction aborts or yields (`retry`), it first reverts any value it has changed based on a per-transaction change log, and then releases all locks it currently holds. The transaction's log is not deleted as it contains information utilized for memoization purposes.

Recall our design supports nested, multi-threaded transactions. A multi-threaded transaction is defined as a transaction whose processing is split among a number of threads. Transactions that perform a collection of operations on disjoint objects can have these operations be performed in parallel. The threads which comprise a multi-threaded transaction must synchronize at the transaction's commit point. Namely, the parent thread will wait at its transaction boundary until its children complete. We allow spawned threads and the parent transaction to communicate through CML message passing primitives. Synchronization invariants among concurrent computation within a transaction must be explicitly maintained by the application. The transaction as a whole, however, is guaranteed to execute atomically with the rest of the computation.

5.2 Memoization

A memo is first created by capturing the procedure's argument at the call site. For each communication within the annotated procedure, we generate a constraint. A constraint is composed of a channel identifier and the value that was sent or received on the channel. In the case of a spawn, we generate a spawn constraint which simply contains the procedure expression which was spawned. Constraints are ordered and augment the parent transaction's log. When a procedure completes, its return value is also added to the log. To support partial memoization, continuations are captured with the generated constraints.

Unlike traditional memoization techniques, it is not readily apparent if a memoized version of a procedure can be utilized at a call site. Not only must the arguments match, but the constraints which were captured must be satisfied in the order they were generated. Thus, we delay a procedure's execution to see if its constraints will be matched. Constraint matching is similar to channel communication in that the delayed procedure will block on each constraint. Constraints can be satisfied either by matching with other constraints or by exchanging and consuming values from channels. Constraints are satisfied if the value passed on the channel matches the value embedded in the constraint. Therefore, constraints ensure that a memoized procedure both receives and sends specific values and synchronizes in a specific order. Constraints make no qualifications about the communicating threads. Thus, a procedure which received a specific value from a given thread may be successfully memoized as long as its constraint can be matched with *some* thread.

If constraint matching fails, pure computation within the application cannot be fully elided. Constraint matching can only fail on a receive constraint. A receive constraint obligates a function to read a specific value from a channel. To match a constraint on a channel with a regular communication event, we are not obligated to remove values on the channel in a specific order. Since channel communication is blocking, a constraint that is being matched can choose from all values whose senders are currently blocked on the channel. This does not violate the semantics of CML since the values blocked on a channel cannot be dependent on one another; in other words, a schedule must

exist where the matched communication occurs prior to the first value blocked on the channel.

Unlike a receive constraint, a send constraint can never fail. CML receives are ambivalent to the value they remove from a channel and thus any receive on a matching channel will satisfy a send constraint. If no receives or sends are enqueued on a constraint's target channel, a re-execution of the function will also block. Therefore, failure to fully discharge constraints by stalling memoization on a presumed unsatisfiable constraint does not compromise global progress. This observation is critical to keeping memoization overheads low.

In the case that a constraint is blocked on a channel that contains no other communications or constraints, memoization induces no overheads, since the thread would have blocked regardless. However, if there exist communications or constraints that simply do not match the value the constraints expects, we can fail, and allow the thread to resume execution from the continuation stored within the constraint. To identify such situations, we have implemented a simple yet effective heuristic. Our implementation records the number of context switches to a thread blocked on a constraint. If this number exceeds a small constant (two in our current implementation), memoization stops, and the thread continues execution within the procedure body at that communication point.

Our memoization technique relies on efficient equality tests for performance and expressivity. We extend MLton's poly-equal function to support equality on reals and closures. Although equality on values of type real is not algebraic, built-in compiler equality functions were sufficient for our needs. To support efficient equality on procedures, we approximate function equality as closure equality. Unique identifiers are associated with every closure and recorded within their environment; runtime equality tests on these identifiers are performed during memoization.

5.3 CML hooks

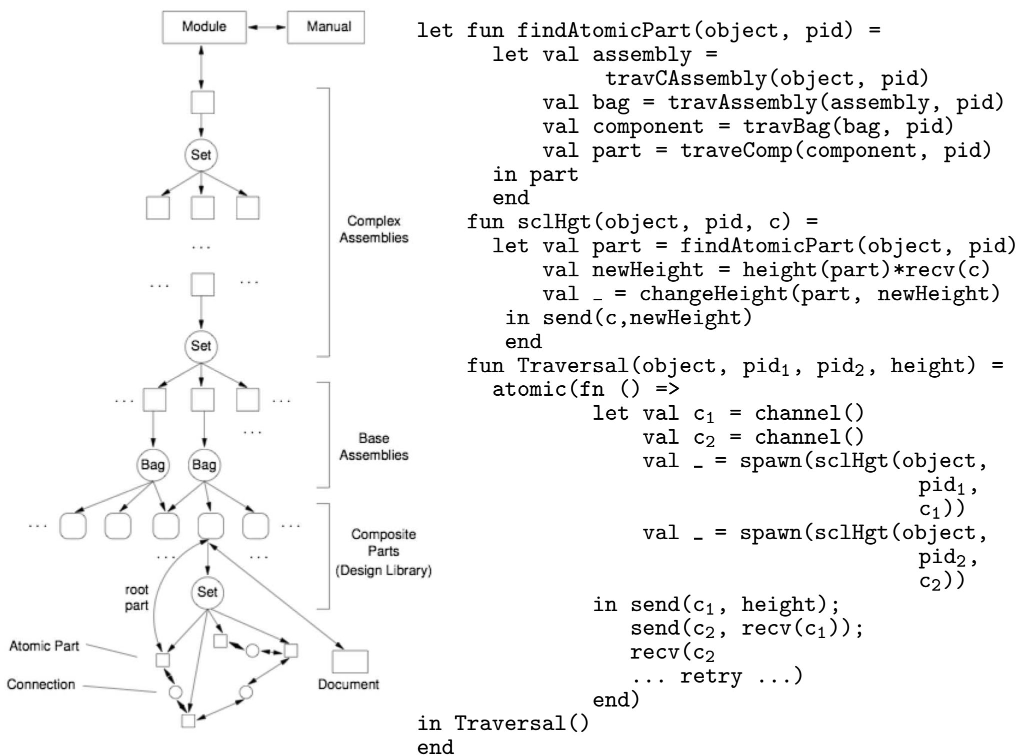
The underlying CML library was also modified to make memoization efficient. The bulk of the changes were hooks to monitor channel communication and spawns, and to support constraint matching on synchronous operations. Successful communications occurring within transactions were added to the log in the form of a constraints, as described previously. Selective communication and complex composed events were also logged upon completion. A complex composed event simply reduces to a sequence of communications that are logged separately.

The constraint matching engine also required a modification to the channel structure. Each channel is augmented with two additional queues to hold send and receive constraints. When a constraint is being tested for satisfiability, the opposite queue is first checked (e.g. a send constraint would check the receive constraint queue). If no match is found, the regular queues are checked for satisfiability. If the constraint cannot be satisfied immediately it is added to the appropriate queue.

6 Case Study - STMBench7

As a realistic case study, we consider STMBench7 [9], a comprehensive, tunable multi-threaded benchmark designed to compare different STM implementations and designs. Based on the well-known 007 database benchmark [5], STMBench7 simulates data storage and access patterns of CAD/CAM applications that operate over complex geometric structures (see Fig. 5).

STMBench7 was originally written in Java. We have implemented a port to Standard ML (roughly 1.5K lines of SML) using our channel based STM. In our implementation, all nodes in the complex assembly structure and atomic parts graph are represented as servers with one receiving channel and handles to all other adjacent nodes. Handles to other nodes are simply the channels themselves. Each server thread waits for a message to be received, performs the requested computation, and then asynchronously sends the subsequent part of the traversal to the next node. A transaction can thus be implemented as a series of channel based communications with various server nodes.




```
let fun findAtomicPart(object, pid) =
      let val assembly =
                travCAssembly(object, pid)
          val bag = travAssembly(assembly, pid)
          val component = travBag(bag, pid)
          val part = traveComp(component, pid)
      in part
      end
    fun sclHgt(object, pid, c) =
      let val part = findAtomicPart(object, pid)
          val newHeight = height(part)*recv(c)
          val _ = changeHeight(part, newHeight)
       in send(c,newHeight)
       end
    fun Traversal(object, pid1, pid2, height) =
      atomic(fn () =>
              let val c1 = channel()
                  val c2 = channel()
                  val _ = spawn(sclHgt(object,
                                       pid1,
                                       c1))
                  val _ = spawn(sclHgt(object,
                                       pid2,
                                       c2))
              in send(c1, height);
                 send(c2, recv(c1));
                 recv(c2
                 ... retry ...)
              end)
in Traversal()
end
```

Fig. 5. The figure on the left shows the overall structure of structure of a CAD/CAM object. The code on the right illustrates a multi-threaded atomic traversal of these objects.

At its core, STMBench7 builds a tree of assemblies whose leafs contain bags of components; these components have a highly connected graph of atomic parts and design documents. Indices allow components, parts, and documents to be accessed via their properties and IDs. Traversals of this graph can begin from the assembly root or any index and sometimes manipulate multiple pieces of data.

The program on the right side of Fig. 5 shows a code snippet that is responsible for modifying the height parameters of a building's structural component. A change made by the procedure `Traversal` affects two components of a design, but the specific changes to each component are disjoint and amenable for concurrent execution. Thus, the modification can easily be expressed as disjoint traversals, expressed by the procedure `findAtomicPart`. The `sclHgt` procedure shown in Fig. 5) changes the height parameter of distinct structural parts. Observe that although the height parameter of $\texttt{pid}_2$ depends on the new height of $\texttt{pid}_1$, the traversal to find the part can be executed in parallel. Once $\texttt{pid}_1$ is updated, the traversal for $\texttt{pid}_2$ can complete.

Consider what would happen if the atomic section is unable to commit. Observe that much of the computation performed within the transaction are graph traversals. Given that most changes are likely to take place on atomic parts, and not on higher-level graph components such as complex or base assemblies, the traversal performed by the re-execution is likely to overlap substantially with the original traversal. Of course, when the transaction executes, it may be that some portion of the graph has changed. Without knowing exactly which part of the graph has been modified by other transactions, the only obvious safe point for re-execution is the beginning of the traversal.

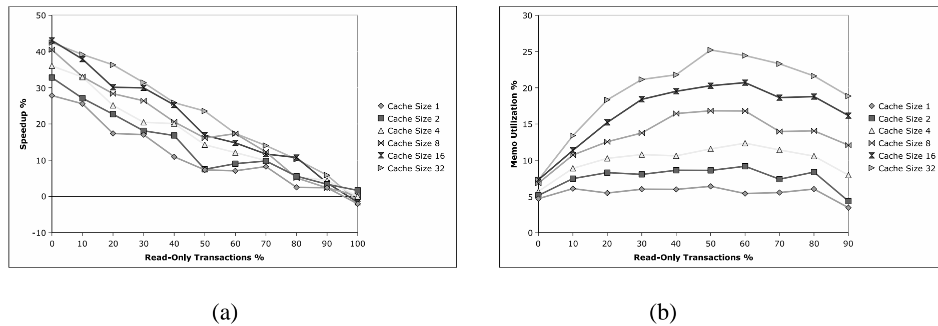
6.1 Results

To measure the effectiveness of our memoization technique, we executed two configurations of the benchmark, and measured overheads and performance by averaging results over ten executions. The *transactional* configuration uses our STM implementation without any memoization. The *memoized transactional* configuration implements partial memoization of aborted transactions. The benchmarks were run on an Intel P4 2.4 GHz machine with one GByte of memory running Gentoo Linux, compiled and executed using MLton release 20051202. Our experiments are not executed on a multiprocessor because the utility of memoization for this benchmark is determined by performance improvement as a function of transaction aborts, and not on raw wallclock speedups.

All tests were measured against a graph of over 1 million nodes. In this graph, there were approximately 280k complex assemblies and 1400K assemblies whose bags referenced one of 100 components; by default, each component contained a parts graph of 100 nodes. Our tests varied two independent variables: the read-only/read-write transaction ratio (see Fig. 6) and part graph size (see Fig. 7). The former is significant because only transactions that modify values can cause aborts. Thus, an execution where all transactions are read-only or which never `retry` cannot be accelerated, but one in which transactions can frequently abort or `retry` offers potential opportunities for memoization. In our experiments, the atomic parts graph (the graph associated with each component) is modified to vary the length of transactions. By varying the number

of atomic parts associated with each component, we significantly alter the number of nodes that each transaction accesses, and thus lengthen or shorten transaction times.

For each test, we varied the maximum number of memos (labeled cache size in the graphs) stored for each procedure. Tests with a small number experienced less memo utilization than those with a large one. Naturally, the larger the size of the cache used to hold memo information, the greater the overhead. In the case of read-only non-aborting transactions (shown in Fig. 6), performance slowdown is correlated to the maximum memo cache size.

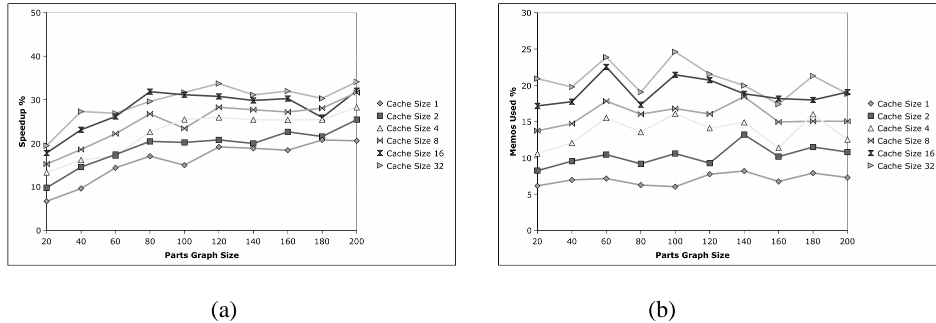


(a) (b)

Fig. 6. (a) presents normalized runtime speedup with a varying read to write ratio. (b) shows the average percent of transactions which are memoizable as read/write ratios change.

Our experiments consider four different performance facets: (a) runtime improvements for transactions with different read-write ratios across different memo cache sizes (Fig. 6(a)); (b) the amount of memoization exhibited by transactions, again across different memo cache sizes (Fig. 6(b)); (c) runtime improvements as a function of transaction length and memo cache size (Fig. 7(a)); and, (d) the degree of memoization utilization as a function of transaction length and memo cache size (Fig. 7). Memory overheads were measured by utilizing MLton's profiler and GC statistics. Memory overheads were proportional to cache sizes and averaged roughly 15% for caches of size 16. Runs with cache sizes of 32 had overheads of 18%.

Memoization leads to substantial performance improvements when aborts are likely to be more frequent. For example, even when the percentage of read-only transactions is 60%, we see a 20% improvement in runtime performance compared to a non-memoizing implementation. The percentage of transactions that utilize memo information is related to the size of the memo cache and the likelihood of the transaction aborting. In cases where abort rates are low, for example when there is a sizable fraction of read-only transactions, memo utilization decreases. This is because a procedure is applied potentially many times, with the majority of applications not requiring memoization because they were not in aborted transactions. Therefore, its memo utilization will be much lower than a procedure in a transaction that aborted once and which was able to leverage memo information when subsequently re-applied.



(a) (b)

Fig. 7. (a) shows normalized runtime speedup compared to varying transactional length. (b) shows the percentage of aborted transactions which are memoizable as transaction duration changes.

To measure the impact of transaction size on performance and utilization, we varied the length of the random traversals in the atomic parts graph. As Fig. 7(a) illustrates, smaller transactions offer a smaller chance for memoization (they are more likely to complete), and thus provide less opportunities for performance gains; larger transactions have a greater chance of taking advantage of memo information. Indeed, we see a roughly 30% performance improvement once the part size becomes greater than 80 when the memo cache size is 16 or 32. As transaction sizes increase, however, the amount of the transaction that is memoizable decreases slightly (Fig. 7(b)). Larger transactions have a higher probability that some part of their traversal has changed and are thus not memoizable. After a certain size, an increase in the traversal length of the atomic parts graph no longer impacts the percent of memos used. This is because the majority of the transaction that is memoizable is found in the initial traversal through the assembly structure, and not in the highly-contented parts components.

As expected, increasing the memoization cache size leads to an increase in both run-time speed up as well as the percent of the transactions that we are able to memoize. Unfortunately, as a result our memoization overheads are also increased both due to the larger amount of memos taken during execution as well as increased time to discover which memo can be utilized at a given call site. Memory overheads increase proportionally to the size of the memo cache.

7 Related Work and Conclusions

Memoization, or function caching [15, 17, 13], is a well understood method to reduce the overheads of function execution. Memoization of functions in a concurrent setting is significantly more difficult and usually highly constrained [6]. We are unaware of any existing techniques or implementations that apply memoization to the problem of optimizing execution for languages that support first-class channels and dynamic thread creation.

Self adjusting mechanisms [2, 3, 1] leverage memoization along with change propagation to automatically alter a program's execution to a change of inputs given an existing execution run. Selective memoization is used to identify parts of the program which

have not changed from the previous execution while change propagation is harnessed to install changed values where memoization cannot be applied. The combination of these techniques has provided an efficient execution model for programs which are executed a number of times in succession with only small variations in their inputs. However, such techniques require an initial and complete run of the program to gather needed memoization and dependency information before they can adjust to input changes.

New proposals [10] have been presented for self adjusting techniques to be applied in a multi-threaded context. However, these proposals impose significant constraints on the programs considered. References and shared data can only be written to once, forcing self adjusting concurrent programs to be meticulously hand crafted. Additionally such techniques provide no support for synchronization between threads nor do they provide the ability to restore to any control point other than the start of the program.

Reppy and Xiao [19] present a program analysis for CML that analyzes communication patterns to optimize message-passing operations. A type-sensitive interprocedural control-flow analysis is used to specialize communication actions to improve performance. While we also use CML as the underlying subject of interest, our memoization formulation is orthogonal to their techniques.

Our memoization technique shares some similarity with transactional events [7]. Transactional events require arbitrary look-ahead in evaluation to determine if a complex composed event can commit. We utilize a similar approach to formalize memo evaluation. Unlike transactional events, which are atomic and must either complete entirely or abort, we are not obligated to discover if an application is completely memoizable. If a memoization constraint cannot be discharged, we can continue normal execution of the function body from the failure point.

References

1. Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An Experimental Analysis of Self-Adjusting Computation. In *PLDI*, pages 96–107, 2006.
2. Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *POPL*, pages 247–259, 2002.
3. Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective Memoization. In *POPL*, pages 14–25, 2003.
4. Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *PLDI*, pages 26–37, 2006.
5. Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The 007 benchmark. *SIGMOD Record*, 22(2):12–21, 1993.
6. Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A Concurrent Logical Framework II: Examples and Applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002.
7. Kevin Donnelly and Matthew Fluet. Transactional Events. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 124–135, 2006.
8. Jim Gray and Andreas Reuter. *Transaction Processing*. Morgan-Kaufmann, 1993.
9. Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: a Benchmark For Software Transactional Memory. In *Eurosys*, 2007.

10. Matthew Hammer, Umut A. Acar, Mohan Rajagopalan, and Anwar Ghuloum. A Proposal for Parallel Self-Adjusting Computation. In *DAMP*, 2007.
11. Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *PPoPP*, pages 48–60, 2005.
12. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software Transactional Memory for Dynamic-Sized Data Structures. In *ACM Conference on Principles of Distributed Computing*, pages 92–101, 2003.
13. Allan Heydon, Roy Levin, and Yuan Yu. Caching Function Calls Using Precise Dependencies. In *PLDI*, pages 311–320, 2000.
14. Richard Kelsey, Jonathan Rees, and Michael Sperber. The Incomplete Scheme 48 Reference Manual for Release 1.1, July 2004.
15. Yanhong A. Liu and Tim Teitelbaum. Caching Intermediate Results for Program Improvement. In *PEPM*, pages 190–201, 1995.
16. MLton. http://www.mlton.org.
17. William Pugh. An Improved Replacement Strategy for Function Caching. In *LFP*, pages 269–276, 1988.
18. William Pugh and Tim Teitelbaum. Incremental Computation via Function Caching. In *POPL*, pages 315–328, 1989.
19. John Reppy and Yingqi Xiao. Specialization of CML Message-Passing Primitives. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 315–326, 2007.
20. John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
21. Michael F. Ringenburg and Dan Grossman. AtomCaml: First-Class Atomicity via Rollback. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 92–104, 2005.
22. Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a High-Performance Software Transactional Memory system for a Multi-Core Runtime. In *PPoPP*, pages 187–197, 2006.
23. Andrew P. Tolmach and Andrew W. Appel. Debuggable Concurrency Extensions for Standard ML. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 120–131, 1991.
24. Lukasz Ziarek, Philip Schatz, and Suresh Jagannathan. Stabilizers: a modular checkpointing abstraction for concurrent functional programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 136–147, 2006.