Compiler and Runtime Support for Efficient Software Transactional Memory

Ali-Reza Adl-Tabatabai[1] Brian T. Lewis[1] Vijay Menon[1] Brian R. Murphy[2] Bratin Saha[1] Tatiana Shpeisman[1]

[1]Intel Labs
Santa Clara, CA 95054

[2]Intel China Research Center
Beijing, China

{ali-reza.adl-tabatabai,brian.t.lewis,vijay.s.menon,brian.r.murphy,bratin.saha,tatiana.shpeisman}@intel.com

Abstract

Programmers have traditionally used locks to synchronize concurrent access to shared data. Lock-based synchronization, however, has well-known pitfalls: using locks for fine-grain synchronization and composing code that already uses locks are both difficult and prone to deadlock. Transactional memory provides an alternate concurrency control mechanism that avoids these pitfalls and significantly eases concurrent programming. Transactional memory language constructs have recently been proposed as extensions to existing languages or included in new concurrent language specifications, opening the door for new compiler optimizations that target the overheads of transactional memory.

This paper presents compiler and runtime optimizations for transactional memory language constructs. We present a high-performance software transactional memory system (STM) integrated into a managed runtime environment. Our system efficiently implements nested transactions that support both composition of transactions and partial roll back. Our JIT compiler is the first to optimize the overheads of STM, and we show novel techniques for enabling JIT optimizations on STM operations. We measure the performance of our optimizations on a 16-way SMP running multi-threaded transactional workloads. Our results show that these techniques enable transactional memory's performance to compete with that of well-tuned synchronization.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization, Run-time environments

General Terms Algorithms, Measurement, Performance, Design, Experimentation, Languages

Keywords Transactional Memory, Synchronization, Locking, Compiler Optimizations, Code Generation, Virtual Machines

1. Introduction

As single thread performance hits the power wall, hardware architects have turned to chip level multiprocessing (CMP) for increasing processor performance. Future processor generations will use the exponentially increasing transistor budget to provide increasing amounts of thread-level parallelism, thus bringing parallel programming into the mainstream. Today, programmers use lock-based synchronization to control concurrent access to shared data. Lock based synchronization is difficult to compose, and can lead to problems such as deadlock. *Transactional memory* (TM) provides an alternate concurrency control mechanism that eliminates or alleviates these pitfalls, and significantly eases parallel programming.

TM can be provided as either a library or a first-class language construct. A library approach allows language designers to experiment with different TM abstractions and implementations before baking TM into a language or hardware. A first-class TM language construct, on the other hand, enables compiler optimizations to improve performance and enables static analyses that provide compile-time safety guarantees. Although several researchers have implemented TM language constructs [16, 33, 17] and language designers have included TM constructs in new concurrent language specifications[2, 11, 9], no prior work has addressed how to support transactions in an optimizing compiler.

In this paper, we focus on compiler and runtime optimizations for transactional memory language constructs. We present a high-performance TM system integrated into a managed runtime environment. We show that a highly tuned software TM application stack performs comparably to well-tuned locking code. Our TM system makes the following novel contributions:

- Our TM implementation is the first TM system to integrate an optimizing JIT compiler with a runtime STM library. We present a new runtime STM interface tailored for JIT-compiled code in a managed environment. We present compiler optimizations to reduce the overhead of STM and show how the compiler intermediate representation can expose STM operations in such a way that existing global optimizations can *safely* eliminate redundant STM operations. Our results show that these optimizations substantially reduce STM overheads to less than 20% over synchronized code on a single thread.
- We are the first STM to provide language constructs for composable memory transactions [17] in an imperative Java-like language. We demonstrate how common transactional idioms (e.g., conditional critical regions) map onto these constructs. Moreover, we demonstrate how to map these constructs onto Java's built-in exception mechanisms in order to model transactional control flow due to nested transactions, user-initiated retry operations, and data conflicts.
- Our TM system is the first to implement and evaluate conflict detection at both object and word granularity; the first to balance conflict detection overhead with conflict detection granu-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'06 June 10–16, 2006, Ottawa, Ontario, Canada

Copyright © 2006 ACM 1-59593-320-4/06/0006...$5.00.

larity by allowing object and word granularity conflict detection to co-exist on a per-type basis; and the first to implement this in a general way that supports a moving garbage collector. Our results show that this strategy is crucial to lowering the overheads of STM while at the same time achieving good scalability when multiple threads concurrently access disjoint elements of a shared object.

- Our TM system efficiently implements nested transactions and partial undos. Our runtime STM structures help implement nesting efficiently by making it easy to merge or discard read and write sets on nested transaction commits and aborts. We show how to model nesting in the intermediate representation so that the compiler can correctly optimize STM operations across nesting depths.

We have implemented our STM in a managed runtime environment that supports multithreaded Java programs employing new transactional language extensions. Our core platform consists of StarJIT[1], a high-performance dynamic compiler for Java and C#, the Open Runtime Platform virtual machine (ORP) [10], and the McRT-STM [34].

We use the Polyglot compiler framework [30] to add transactions to the Java language and to encode them into standard Java. We have extended the StarJIT intermediate representation (STIR) to include STM operations and modified existing optimizations to transform these operations correctly. Additionally, we implemented new optimizations that specifically target STM operations. The executable code generated by StarJIT directly invokes the McRT-STM library embedded in ORP. We have tailored and optimized this STM for use in a managed runtime environment and for use as a JIT compiler target.

We measure the performance and scalability of our system on a 16 processor SMP system using a set of Java workloads rewritten to use transactions instead of lock-based synchronization. Our results show that our techniques enable the performance of transactional memory to compete with the performance of well-tuned synchronization.

The rest of this paper is organized as follows. Section 2 describes our transactional extensions to Java. Section 3 describes our software transactional memory implementation. Section 4 describes the transactional memory support and optimizations in StarJIT. Section 5 presents our experimental results, and Section 6 discusses related work.

2. Transactions in Java

To support program-level transactions, we defined several new constructs as extensions to the Java language. First, we added an atomic construct of the form `atomic{S}` that executes the statement `S` as a transaction (the same basic `atomic` construct as [16, 2, 9, 11]). If a thread executes a statement `S` atomically, all memory effects in `S` are serializable with respect to transactions in other threads. When `S` completes (either normally or exceptionally), its effects become globally visible.

Additionally, we added two operations that provide composable memory transactions[17]: `retry` and `orelse`. A `retry` operation, which may be invoked only inside a transaction, blocks a thread and restarts its transaction when an alternate path is available. It indicates that the transaction cannot complete with the current state of shared memory and must instead wait for another thread's transaction to alter the memory locations that it depends on. Similar to Java's `wait` operation, it provides a facility for condition synchronization. In contrast, there is no corresponding `notify` operation; notification is implicitly handled by our system.

An `orelse` operation composes two or more alternative transactions. For example, the operation `atomic{S1}orelse{S2}` first executes `S1` as a nested transaction. If `S1` completes, the operation is complete. If it blocks using `retry`, its side effects (including those to local variables) are discarded, and `S2` is executed. An `orelse` operation requires support for nested transactions in order to partially undo the effects of its clauses. Moreover, in our system, `atomic` and `atomic-orelse` operations may be arbitrarily nested and composed. To ensure the latter, we permit a `retry` operation to block only an outer-level transaction.

```
                      atomic {
tryatomic {             S;
  S;          →       } orelse {
}                       throw new TransactionFailed();
                      }
```

```
when(cond) {          atomic {
  S;          →         if(!cond)
}                         retry;
                        S;
                      }
```

Figure 1. Fortress-style `tryatomic` and X10-style `when`.

```
while(true) {
  TxnHandle th = txnStart();
  try     { S; break; }
  finally { if(!txnCommit(th)) continue;}
}
```

Figure 2. Java code generated for `atomic{S}`.

These three constructs allow our system to support an interesting subset of the transactional features discussed in the literature. To illustrate this, we also support two additional constructs in terms of the first three. A `tryatomic` operation, similar to that in Fortress [2], attempts to execute a statement atomically, but immediately fails with an exception if it tries to block. A `when` operation, similar to that in X10 [9] and a conditional critical region (CCR) in [16], blocks until a condition holds and then atomically executes the corresponding statement. As shown in Figure 1, both `tryatomic` and `when` are syntactic sugar for the corresponding sequences on the right.

With these constructs, we do not seek to define a complete, formal extension to the Java language. Instead, we intend to study transactional constructs within the context of Java and similar languages. As such, certain issues that require robust solutions for completeness are beyond the scope of this paper: (1) We disallow native method invocations within a transaction, unless they are standard Java library methods that are known to be side-effect free. Any other native method invocations cause a Java error to be thrown. (2) We do not explore interactions between transactions and Java's built-in synchronization features such as `wait` and `notify` operations. Other work has explored how such constructs may interact with or be implemented in terms of transactions, but we do not address this further here. (3) We assume that all potentially concurrent accesses to shared memory are properly guarded by atomic regions. Any accesses that violate this are not serialized with respect to the atomic regions and may result in invalid and unexpected values. This is consistent with the Java memory model's treatment [23] of synchronized regions. In the future, we plan to explore a *strongly atomic* [6] system in which every memory access is implicitly atomic.

We implemented our Java extensions using the Polyglot compiler toolkit [30]. We use Polyglot to compile a transactional Java program to a pure Java program that is in turn compiled to class

```
while(true) {
  TxnHandle th = txnStart();
  try {
    while(true) {
      TxnHandle th1 = txnStartNested();
      try { S1; break; }
      catch(TransactionRetryException tex) {
        txnAbortNested(th1);
      }
      S2; break;
    }
    break;
  }
  finally { if(!txnCommit(th)) continue; }
}
```

Figure 3. Java code generated for `atomic{S1}orelse{S2}`.

files. Polyglot translates the new constructs to Java library calls that StarJIT recognizes and uses Java's exception handling to implement the control flow due to transaction aborts and retries.

Figure 2 shows the Java code Polyglot generates for `atomic{S}`. This code invokes `txnStart` to begin the transaction and to define a corresponding *transaction handle* `th`. The transaction handle represents a particular transaction nesting scope. As we show in Section 4, StarJIT uses the handles to qualify the scope of the STM operations that it generates. Although it is not shown here, the statement `S` is also recursively modified by Polyglot. Any `atomic` operations within `S` are modified accordingly. Any `retry` operation is mapped directly to a `txnUserRetry` call, which will first set the internal STM state to indicate an in-progress retry and then throw a `TransactionRetryException` object. Finally, after `S` is executed, it invokes `txnCommit` to end the transaction on any exit path (normal, exceptional, or retry) from `S`. The `txnCommit` call attempts to commit the transaction. If it succeeds, it returns true; the transaction completes, and control flow exits the loop. If it fails, it aborts the transaction and returns false; the loop continues, and the transaction restarts.

In general, the nesting of transactions is ambiguous in the generated Java code; a `txnStart` or `txnCommit` may be nested within another transaction at runtime. To support composable blocking, the semantics of `txnCommit` depends on whether it is dynamically nested. An outer-level `txnCommit` (one that is not dynamically nested) will check the internal state for a retry, and if it occurred, it will clear the state, discard side effects, block on the read set (as discussed in Section 3), and return false, restarting the transaction. On the other hand, a nested `txnCommit` will return true on a retry, propagating it to an outer transaction.

Figure 3 shows the Java code Polyglot generates for `atomic {S1} orelse {S2}`. Intuitively, an `atomic-orelse` signifies a single transaction with two or more nested transactions — one for each clause[1]. The code reflecting the outer transaction is identical to that described above. The code for the nested transactions, `S1` and `S2`, appear within a second while loop. To begin a nested transaction, the code invokes `txnStartNested`. If the nested transaction completes normally, control exits the inner loop. No further nested transactions are executed. If the transaction triggers an exception (other than a retry), the exception also cascades beyond the inner loop. On the other hand, if a retry is encountered, a `TransactionRetryException` will be caught, `txnAbortNested` will undo the side effects of the nested transaction, and control will transfer to the next nested transaction. If the final nested transaction triggers a retry, we cascade the `TransactionRetryException` outside the loop to the next outer transaction (which may, in turn, have alternatives). As shown in Figure 3, we flatten the last nested transaction as an optimization because its effects are guaranteed to be undone by an outer transaction on a retry.

We have elided a couple important details in Figures 2 and 3. First, we restore local variables when a transaction aborts. On an abort, Polyglot inserts code to restore the value of any local variables that were modified by the transaction and are still live. Second, we must ensure that user exception code does not interfere with our STM exception classes. Logically, `TransactionRetryException` and other internal STM exception classes are outside the standard `java.lang.Throwable` exception hierarchy. We have modified StarJIT and ORP accordingly to ensure that `finally` and `catch` clauses in user code do not execute for STM exceptions. The `finally` and `catch` clauses generated by Polyglot (e.g., in Figures 2 and 3) do execute for STM exceptions to support transactional control flow, however.

After we run Polyglot, we use a standard Java compiler to generate bytecode with STM calls. At execution time, the ORP virtual machine loads these class files and invokes StarJIT to compile them. StarJIT converts the STM library calls into STIR operations and inserts additional operations to support the underlying STM implementation.

As in [16], the ORP VM creates (on demand) transactional clones of methods called within a transaction. The original method is guaranteed to be invoked only outside a transaction, while the clone is guaranteed to be invoked only inside a transaction. During compilation, StarJIT knows whether it is compiling the original or the cloned version of a method. If a method call occurs within a transaction, StarJIT will redirect it to invoke the transactional clone instead.

StarJIT maps nested `txnStart` and `txnCommit` operations to `txnStartNested` and `txnCommitNested` operations, respectively. It optionally flattens such nested transactions (removing them altogether) to eliminate their overhead. Nested transactions limit optimization of undo logging operations across nesting depths (which we explore in Section 4) so flattening exposes optimization opportunities. Flattening, however, precludes partial rollback on data conflicts.

3. STM implementation

Our STM [34] implements strict two-phase locking[14] for writes and optimistic concurrency control using versioning[22] for reads. In contrast to object-cloning and write buffering schemes, our STM updates memory locations in place on writes, logging the location's original value in an *undo log* in case it needs to roll back side effects on an abort.

Our scheme has a number of benefits. First, it allows fast memory access inside transactions: an access can go directly to memory without the indirection imposed by a wrapper object and without a check to see if the transaction has previously written the location (i.e., handling read-after-write dependencies is trivial). Second, it avoids the expense of cloning large objects (e.g., arrays). Finally, it allows the programmer to use existing libraries inside transactions because it does not need special wrapper types.

Figure 4 shows the interface that our STM provides to JIT compiled native code. The first six functions correspond to the same functions generated by Polyglot; StarJIT maps the Polyglot-generated calls directly onto these functions. The rest of these functions track transactional memory references and are introduced into the code by StarJIT.

Every transaction maintains transaction-related metadata in a thread-local structure called the *transaction descriptor*. The STM maintains a pointer to the transaction descriptor in thread local storage. On IA-32 platforms, the transaction descriptor can be

[1] We allow an arbitrary number of clauses: `atomic {S1} orelse {S2} orelse ... orelse {Sn}`.

```
void txnStart();
bool txnCommit();
void txnStartNested(TxnMemento*);
bool txnCommitNested(TxnMemento*);
void txnAbortNested(TxnMemento*);
void txnUserRetry()
     throws TransactionRetryException;
void txnValidate()
    throws TransactionException;
void txnOpenObjectForRead(Object*)
    throws TransactionException;
void txnOpenObjectForWrite(Object*)
    throws TransactionException;
void txnOpenClassForRead(ClassHandle*)
    throws TransactionException;
void txnOpenClassForWrite(ClassHandle*)
    throws TransactionException;
void txnOpenWordForRead(Object*,int offset)
    throws TransactionException;
void txnOpenWordForWrite(Object*,int offset)
    throws TransactionException;
void txnLogObjectRef(Object*,int offset);
void txnLogObjectInt(Object*,int offset);
void txnLogStaticRef(void* addr);
void txnLogStaticInt(void* addr);
void txnHandleContention(TxnRecord*)
    throws TransactionException;
void* txnHandleBufOverflow(void* bufPtr);
```

Figure 4. Runtime STM interface for JIT-compiled code.

retrieved with a single load instruction using the `fs` register on Windows or the `gs` register on Linux.

A pointer-sized *transaction record* (`TxnRecord`) tracks the state of each object or memory word accessed inside a transaction. The transaction record can be in one of two states: (1) *shared*, which allows read-only access by any number of transactions, or (2) *exclusive*, which allows read-write access by only the single transaction that owns the record. In the shared state, the record contains a version number, while in the exclusive state it contains a pointer to the owning transaction's descriptor. The least-significant 2 bits of the record encode its state: 00 indicates that the transaction record points to a transaction descriptor, and 01 indicates that the transaction record contains a version number (version numbers are odd and descriptor pointers are 4-byte aligned).

Our STM keeps the mapping between memory locations and transaction records flexible, allowing conflict detection at either the object or word granularity. Object granularity signals a conflict between two transactions that access the same object O if at least one of them writes a field of O, even if they access disjoint fields. Object granularity conflict detection may therefore limit concurrency on large objects — for example, a transaction that writes to an element of an array A will conflict with all other transactions that access disjoint elements of A. Word granularity conflict detection enables finer grain concurrency by allowing concurrent access to different fields of the same object, but has a higher overhead because it requires a check at each memory access. Our STM allows object and word granularity conflict detection to coexist on a per-type basis — for example, arrays can use word-based conflict detection while non-array objects use object-based conflict detection.

Each heap object has an extra slot that holds either a transaction record or a hash value initialized at object allocation time. For object-level conflict detection, the STM uses the slot as the object's transaction record. For word-based conflict detection, the STM uses the slot as a hash value and combines it with the offset of an accessed field or element to form a hashed index into a global table of transaction records. This table keeps each transaction record in a separate cache line to avoid false sharing across multiple processors. Each class structure has a similar slot for detecting conflicts on static field accesses.[2]

3.1 Tracking transactional memory accesses

To track transactional memory accesses, the STM maintains for each transaction a read set, a write set, and an undo log. The read and write sets each consist of a vector of tuples $\langle T_i, N_i \rangle$, where T_i points to a transaction record and N_i is the version number T_i held when the tuple was created. The undo log consists of a vector of tuples $\langle A_i, O_i, V_i, K_i \rangle$, where A_i points to a field or element the transaction updated, O_i refers to the object that contains A_i (null if A_i is a static field), V_i is the value A_i held at the time the tuple was created, and K_i is a tag indicating whether V_i is a reference type. Our current implementation runs on 32-bit machines, so V_i is a 32-bit value, and the JIT breaks up 64-bit values into two undo log entries.

A transaction must *open* an object or memory word (depending on the conflict detection granularity) for reading or writing before accessing it. The JIT inserts the open for read and write operations automatically — selecting between word- and object-level conflict detection granularity based on the accessed object's type — and optimizes these operations as described in Section 4. The JIT also inserts and optimizes undo logging operations before field and array element stores. Note that in our STM, the granularities of conflict detection and undo logging are independent: conflict detection occurs at either the object or word level, while undo logging always occurs on 32-bit values.

The open for read functions, `txnOpenObjectForRead`, `txnOpenWordForRead`, and `txnOpenClassForRead`, first load the transaction record address of the opened object, word, or class, and then call the internal STM function `txnOpenTxnRecordForRead`. The corresponding open for write functions do the same but call `txnOpenTxnRecordForWrite`.

Figure 5 shows the algorithms for `txnOpenTxnRecordForRead` and `txnOpenTxnRecordForWrite`, which open a transaction record for reading and writing, respectively. If the transaction already owns the transaction record, then these functions simply return. In the case of a read, the transaction checks whether the transaction record contains a version number, logs it into the read set, and returns. In the case of a write, the transaction takes ownership of the record before logging into the write set. In either case, if a different transaction owns the transaction record, then `txnHandleContention` is called, which invokes the contention manager.

The contention manager uses randomized exponential backoff, and can use the size of the aborted transaction's read or write set to parameterize the backoff. The contention manager can also implement a host of different contention policies [20]. The contention manager avoids deadlock by using timeouts: it aborts the transaction if the thread owning a transaction record R takes longer than a threshold to release ownerships of R. To abort, the contention manager sets the internal transaction descriptor state and throws a `TransactionException`. This exception eventually unwinds to the outermost `txnCommit` operation, which rolls back the transaction and backs off.

There are four functions that log the original value of a memory location before it is overwritten by a store. The `txnLogObjectRef` function logs the value of a field or array element that holds an object reference, and `txnLogObjectInt` logs a 32-bit heap location

[2] Alternatively, an implementation could easily provide one transaction record slot per static field, allowing concurrent write access to disjoint static fields of the same class.

```
void txnOpenTxnRecordForRead(TxnRecord* rec) {
  void* recValue = *rec;
  void* txnDesc = getTxnDesc();
  if (recValue == txnDesc) return;
  do {
     if (isVersion(recValue)) {
        logReadSet(rec, recValue);
        return;
     }
     txnHandleContention(rec);
     recValue = *rec;
  } while (1);
}

void txnOpenTxnRecordForWrite(TxnRecord* rec) {
  void* recValue = *rec;
  void* txnDesc = getTxnDesc();
  if (recValue == txnDesc) return;
  do {
    if (isVersion(recValue)) {
      if (CAS(rec, recValue, txnDesc)) {
         logWriteSet(rec, recValue);
         return;
      }
    }
    txnHandleContention(rec);
    recValue = *rec;
  } while (1);
}
```

Figure 5. Open for read and write algorithms. The function `CAS` performs an atomic compare-and-swap instruction and returns `True` if the compare and swap succeeds.

that does not hold an object reference. The `txnLogStaticRef` and `txnLogStaticInt` functions do the same but for static fields.

The runtime organization of the read set, write set, and undo log is highly tuned to support frequent STM operations efficiently. These sets are organized as sequential buffers similar to the sequential store buffers used by some generational garbage collectors to hold updates to reference fields [5]. The transaction descriptor contains pointers to the beginning and frontier of these buffers. Appending an entry to a buffer involves an overflow check followed by an increment of the frontier pointer. To make the overflow check fast, each buffer is 2^N bytes and aligned on a 2^N byte address. When a frontier pointer overflows, a new buffer is allocated and linked to the existing buffer.

The sequential buffers make commit and garbage collection root set enumeration very fast because they allow quick iteration over the read and write sets, as well as the undo log.

3.2 Garbage collection support

Because the read set, write set, and undo log may contain references to heap locations, the STM enumerates their contents to the garbage collector (GC) as part of the root set, and the GC updates the contents when it moves objects. To reduce coupling within our system, the STM uses a GC callback API to enumerate potential heap references inside the sequential buffers; the GC is mostly unaware of the STM. The ORP GC filters out references to non-heap transaction records in the read and write sets during enumeration. Because undo log entries may point to the interior of heap objects, the undo log maintains pointers to the base of heap objects. The STM avoids enumerating non-reference values using the undo log entry tags.

Our scheme for mapping memory locations and objects to transaction records allows the garbage collector to move objects without affecting the mapping — it does not preclude a moving GC and it does not require the GC to abort transactions when it moves objects. In contrast, the hashing scheme in [16] uses object addresses, which change and thus invalidate the mapping when the garbage collector moves objects.

3.3 Transaction start and commit

On start, `txnStart` simply sets the transaction state to active and resets the sequential buffers. On commit, `txnCommit` executes the steps shown in Figure 6. First, it validates the read set: it checks that, for each pair $\langle T_i, N_i\rangle$ in the read set, either the current value held in T_i equals N_i, or there exists a tuple $\langle T_j, N_j\rangle$ in the write set such that $T_i = T_j$ and $N_i = N_j$ (i.e., the current transaction owns T_i and the version of T_i at the point the transaction acquired ownership equals N_i). Second, if validation fails or the transaction is not in the active state (due to a user retry or a failed open operation), `txnCommit` invokes `txnAbortTransaction` to abort the transaction and returns false to restart it (as shown in Figure 2). Finally, if the transaction is in the active state and its read set is valid, it changes the state to commit the transaction and releases ownership of each transaction record in the write set by incrementing their version numbers: for each $\langle T_i, N_i\rangle$ in its write set, it sets the current value of T_i to $N_i + 4$.

On abort, `txnAbortTransaction` rolls back the transaction and invokes the contention manager. If the abort was due to a validation failure or a failed open, the contention manager will backoff as described earlier. If the abort was due to a user retry, the contention manager will also block until another thread invalidates the read set; this is a necessary condition in order for the transaction to proceed on a different path [17]. More precisely, `txnAbortTransaction` executes the following steps. First, it restores the original values of the memory locations written by the transaction using the undo log: for each $\langle A_i, O_i, V_i, K_i\rangle$ in its undo log starting from the most recent entry, it sets the value of A_i to V_i. Second, if the transaction is in the retry state (due to a user retry), it preserves the read set: it copies the contents of the write set into the read set [3], and it increments (by 4) the version number for all tuples in the read set where the transaction record is owned by the current transaction. Third, it releases ownership of the transaction records in the write set by incrementing their original version number: for each $\langle T_i, N_i\rangle$ in its write set, it sets the current value of T_i to $N_i + 4$. Finally, it invokes the contention manager as described above: if the transaction is in the retry state, it waits (by spinning with exponential backoff and scheduler yields) until another thread invalidates the read set before returning.

To prevent the program from looping indefinitely due to an inconsistent view of memory, the JIT inserts calls to `txnValidate` (which validates the read set by calling `txnValidateReadSet`) on backward branches and tries to optimize these away for loops that have constant bounds. Alternatively, the STM can use scheduler hooks and validate a transaction at a time slice. Note that any errors (e.g., runtime exceptions) due to an inconsistent view of memory will eventually be caught by `txnValidateReadSet` before the transaction completes via either a JIT inserted call or an explicit call in `txnCommit`.

Our current implementation uses 32 bit version numbers, but an implementation may choose to use 64 bit version numbers (or try to use additional bits from the object header) to guard against version numbers rolling over, and causing validation to return an incorrect result. Note that the probability of an erroneous validation due to 32 bit version numbers is extremely small. For a validation error, a

[3] Our system may elide open for read operations that follow open for writes. Thus, we must conservatively assume that any tuple in the write set may also represent a read value that affected the control flow.

```
bool txnCommit() {
  void* txnDesc = getTxnDesc();
  if ((txnValidateReadSet() == False) ||
      (CAS(&txnDesc->state, Active, Commit) == False)) {
     txnAbortTransaction(); // abort transaction
     return False;
  }
  for <txnrec, ver> in transaction's write set {
      *txnrec = ver + 4; // release ownership
  }
  return True;
}

bool txnValidateReadSet() {
 void* txnDesc = getTxnDesc();
 for <txnrec, ver> in transaction's read set {
   currentVersion = *txnrec;
   if (currentVersion == ver)
      continue;
   if (currentVersion == txnDesc &&
       <txnrec, ver> in transaction's write set)
       continue;
   return False; // validation failed
 }
 return True;
}
```

Figure 6. Commit and validation algorithms.

transaction record's version number needs to roll over and assume exactly the same value at the end of a transaction as its value when the record was read during the transaction.

3.4 Nested transactions

Our STM implements *closed* nested transactions [29] and supports both (1) composable memory transactions and (2) partial aborts when a data conflict occurs within a nested transaction. Our implementation takes advantage of the static block structure of the atomic construct by saving the state required for a nested transaction in the runtime stack of activation records, and uses the Java exception mechanism to unwind the stack and rollback nested transactions. The stack mechanism also allows us to implement partial rollbacks on validation failure or contention.

For each nested transaction, StarJIT allocates a *transaction memento* structure (`TxnMemento`) in the activation record. The transaction memento records the frontier pointers of the transaction descriptor's read set, write set, and undo log at the start of the nested transaction. The STM function `txnStartNested` starts a nested transaction and copies the three buffer pointers from the transaction descriptor into the memento.

Nested transactions are very lightweight when they commit: a nested transaction incurs only the overhead of saving the three buffer pointers into its memento in the common case.

The `txnAbortNested` function undoes the writes performed by the nested transaction. It first restores the values logged in the undo log buffer starting from the last undo log buffer entry up to the entry recorded in the memento. It then resets the transaction descriptor's undo log frontier pointer to the value recorded in the memento. It does not reset the read set or write set frontier pointer, effectively merging the nested transaction's read set and write set with its parent transaction's read set and write set respectively. This is necessary for implementing nested transaction retries for composable memory transactions; the system must still record all memory locations that affected the control-flow in case the program blocks. It must preserve the nested write set as well, as our system may have elided open for read operations that followed open for writes.

```
a.x = 0;
c.y = null;
atomic {
    S1: a.x = 1;
    atomic {
        S2: c.y = b.y;
        ...
    }
    ...
}
```

Figure 7. Example of a simple nested transaction

The memento structures allow us to rollback a transaction partially on a read set validation failure. To do this, read set validation starts validation from the first entry in the read set, and records in the transaction descriptor the first element k containing an invalid version number. During the stack unwinding propagating `TransactionException`, each `txnCommitNested` operation checks its memento to find the first memento M whose read set pointer occurs before k in the read set. Since the validation error occurred within M's transaction, the transaction can abort and retry M rather than aborting and restarting at the top level transaction. To perform a partial abort, `txnCommitNested` first performs the same actions as `txnAbortNested` and then resets the transaction descriptor's read set frontier pointer to the value recorded in the memento.

3.5 Example

We illustrate the data structures described in this section using the code fragment in Figure 7. This example involves one transaction nested inside another. The top-level transaction writes field `x` of object `a` (statement S1). The nested transaction copies field `y` from object `b` to object `c` (statement S2).

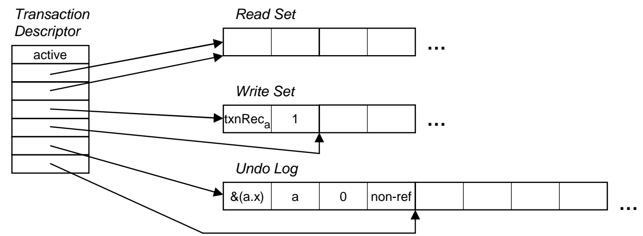

Figure 8. STM data structures after executing statement S1 in Figure 7.

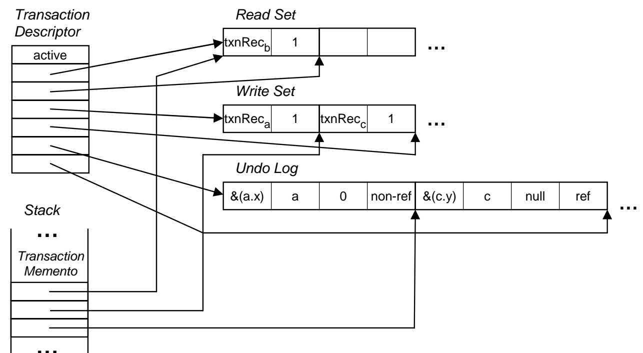

Figure 9. STM data structures after executing statement S2 in Figure 7.

Figure 8 illustrates the state of our STM data structures after executing statement S1. The transaction descriptor contains the transaction state (active) and pointers to the beginning and frontier of the three buffers: the read set, the write set, and the undo log. At this point, the read set is empty as the transaction has not yet opened any objects for read.[4] The write set contains a single entry with the transaction record ($txnRec_a$) and version number (1) of the written object a. The undo log has an entry for the field a.x that specifies the address of the written field (&(a.x)), the object containing the field (a), the old value of the field (0), and a tag indicating that a.x is not a reference field (non-ref).

Figure 9 shows the state of the STM data structures after executing statement S2 inside the nested transaction. At the start of the nested transaction, our STM copies the frontier pointers from the transaction descriptor into a transaction memento allocated on the run-time stack. It then proceeds with the execution of the statement S2: it opens object b for read, it reads the value of b.y, it opens object c for write, it records the old value of c.y in the undo log, and finally, it sets c.y to the new value. Figure 9 reflects the changes to the read set, write set, and undo log due to these actions: a read set entry for object b, a write set entry for object c, and an undo log entry for field c.y.

4. Compiler optimizations for STM

StarJIT performs three types of optimizations targeting STM overheads: (1) optimizations that leverage the existing SSA-based global optimization phases, (2) new transaction-specific global optimizations, and (3) transaction-specific code generation for IA32. This section covers these optimizations.

4.1 Leveraging existing optimizations

The initially-generated STIR representation contains many opportunities for optimization. In particular, the STM operations are amenable to many conventional optimizations already performed by StarJIT such as redundancy elimination, dead code elimination, inlining, and loop transformations. Consider, for example, the code in Figure 10. The first atomic block (labeled A1) contains four writes to the object referenced by the obj variable. Assuming object-level conflict detection, straightforward code generation for the first atomic block produces four open for write operations on obj, of which only the first is necessary. Furthermore, each STM operation inside function foo must get the transaction descriptor held in thread-local storage; it's sufficient to get that descriptor just once. Also, control-flow transformations such as loop peeling and inlining increase redundancy elimination opportunities.

We carefully designed our representation of STM operations in such a way that existing optimization phases can *safely* eliminate STM operations with virtually no modifications. In particular, transactions impose two safety constraints on optimizations that we encode in STIR:

1. *Transaction scope* constraints ensure that STM operations associate with the correct transaction scope in the IR. In Figure 10, for example, optimizations cannot eliminate the open for write required for statement S5 even though it is dominated by other open for write operations in a different transaction. Similarly, even though optimizations can eliminate statement S2's open for write operation because it is dominated by the same open for write at statement S1, S2 still requires an undo logging operation because A1_1 may abort and rollback (e.g., due to a retry).

[4] In this example, we assume object-level contention. With word-level contention, the data structures would be identical except that the read and write sets would contain different (word-level) transaction records.

```
foo() {
    ...
    A1: atomic {
        S1: obj.f1 = x1;
        ...
        A1_1: atomic {
            S2: obj.f1 = x2;
            S3: obj.f2 = x3;
        } orelse {...}
        ...
        S4: obj.f2 = x4;
    }

    A2: atomic {
        S5: obj.f3 = x5;
    }
}
```

Figure 10. Nested transaction source example.

2. *Dependence* constraints explicitly connect open and log operations with the memory accesses they guard. These constraints serve two purposes. First, they enable dead code elimination; for example, if a loaded value becomes dead or a store operation becomes unreachable, the corresponding open or log operations are unnecessary if they guard no other operations. Second, these dependences ensure that code motion optimizations such as instruction scheduling do not move a memory access operation before the STM operations that guard it.

To satisfy the transaction scope constraints, StarJIT uses the transaction handles. STIR STM operations take a transaction handle operand (as well as a transaction descriptor operand). The transaction handle acts as a qualifier on an operation so that the scope of an operation is syntactically known. Commonly used redundancy elimination algorithms typically rely on either lexical or value equivalence [7], and the transaction handle suffices for both.

In the presence of nesting, multiple transaction handles may be in scope. It is always safe to use the handle corresponding to the innermost transaction. As discussed above, open operations are redundant within the same outer-level transaction; as an optimization, we use the outermost handle that is in scope for these operations.

To satisfy the second constraint, StarJIT uses *proof variables*[26] to explicitly represent safety dependences. The undo logging and open operations generate proof variables that are consumed by the corresponding load and store operations. These proof variables simplify dead code elimination. If a proof variable becomes dead during the course of optimization, the open or logging operation that produces it may be safely removed. Furthermore, they constrain code motion optimizations such that transformations cannot move load and store operations above their corresponding open operations, and cannot move write operations above their corresponding undo log operations. Finally, the StarJIT type checker verifies that these proof variables are used correctly, assisting us in the process of developing and debugging optimizations.

In addition to these changes, we made other STM-specific modifications to existing optimizations. We modified code motion to hoist transaction descriptor loads out of loops to reduce the thread local storage accesses that it entails. We modified our dominator-based common subexpression elimination (based on [7]) to eliminate open for read operations that are dominated by open for write operations. Finally, we modified the inliner to more aggressively inline transactional methods and, thus, introduce further optimization opportunities.

4.2 Transaction-specific global optimizations

In addition to leveraging existing StarJIT optimizations, we implemented several new STM specific optimizations. The first optimization eliminates STM operations for accesses to provably immutable memory locations. In Java, final fields may only be defined once, in the class initializer for static fields or in the object constructor for instance fields. Once the class or object is initialized, the final field is immutable [23] and no open for read operation is necessary to access it. StarJIT suppresses generation of an open for read operation for accesses to final fields (including accesses to the vtable slot for type checks and method dispatch, and accesses to the array length field). Some standard library types, such as the `String` class in the `java.lang` package, are known to be immutable. For method invocation bytecodes that invoke safe methods belonging to one of these immutable types, StarJIT generates code that invokes the non-transactional versions of the methods. Additional immutable fields could be discovered via whole program analysis; however, we did not pursue this as it is often impractical in production systems that permit dynamic class loading.

The second optimization suppresses STM operations for transaction local objects. Objects allocated inside a transaction do not need to be opened before accessing in the same transaction, as, by atomic semantics, they will not be visible to other threads. In the presence of nesting, StarJIT only suppresses log operations if the object was allocated within the same transaction or a nested transaction. A nested transaction needs to generate an undo log call if it stores to an object allocated in a parent transaction (but it doesn't have to open it). Similarly, transactional versions of constructors do not need to open the this pointer and do not require undo logging except inside any nested transaction inside the constructor.

To expose more redundancy elimination opportunities, StarJIT transforms a program to proactively acquire write locks in certain cases. When an open for read dominates an open for write operation with matching arguments (e.g., the same object within the same transaction) but profile information suggests that the open for write operation will executed almost as frequently, StarJIT replaces the open for read with an open for write. Subsequently, any following open for write operations will be eliminated.

4.3 Transaction-specific code generation

Finally, we implemented partial inlining of frequent STM library calls to reduce overhead and to expose further compiler opportunities. The sequential buffers that our STM operations use lead to very efficient instruction sequences, illustrated in Figure 11. This figure shows the IA32 assembly code for opening a transaction record for writing. (The code for opening for read is the same as this except that it omits instructions `I5` – `I6`, and it uses the read set buffer pointer offset.) The code is organized into a hot portion containing the common path, and an out-of-line cold portion (starting at instruction `I14`) containing the less frequent case of buffer overflow (handled by `txnHandleBufOverflow`) and the slow case of opening a transaction record owned by another thread (handled by `txnHandleContention`). We can shorten the hot path even further by moving instructions `I1` and `I2` — which check whether a transaction is accessing a transaction record that it already owns — into the cold portion if this case occurs infrequently. We can also eliminate the overflow check instructions (`I8` – `I9`) using a guard page after each buffer, and relying on page protection hardware and OS signal handling.

The JIT inlines the code sequence of Figure 11 to avoid the branching and register save-restore overheads of a function call in the common case and to promote the buffer pointer into a register inside loops (eliminating `I7` & `I11`).[5] The measurements in Section 5 show that this improves performance significantly. Like the open for read and write sequences, the undo logging sequence is also very efficient and can be optimized via inlining.

```
I0:   mov eax, [txnrecord]
I1:   cmp eax, txndesc
I2:   jeq done   // this txn owns txnrecord
I3:   test eax, #versionmask
I4:   jz contention
I5:   cmpxchg [txnrecord],txndesc
I6:   jnz contention
I7:   mov ecx, [txndesc+wsbufptr]
I8:   test ecx, #overflowmask
I9:   jz overflow
I10:  add ecx, 8
I11:  mov [txndesc+wsbufptr],ecx
I12:  mov [ecx-8], txnrecord
I13:  mov [ecx-4], eax
done:
      . . .
contention: // handle contention
I14:  push txnrecord
I15:  call txnHandleContention
I16:  jmp I0
overflow: // handle buffer overflow
I17:  push ecx
I18:  call txnHandleBufOverflow
I19:  jmp I0
```

Figure 11. Code to open a transaction record for writing. The `txnrecord` and `txndesc` registers point to the transaction record and the transaction descriptor, respectively. `wsbufptr` is the offset of the write set buffer pointer in the transaction descriptor.

5. Experimental results

We evaluated our STM system against Java's built-in synchronization facilities. We measured the sequential overhead it imposes on a single thread, and its scalability on a 16-processor machine. In this section, we demonstrate that transactional Java programs running on our STM are competitive with synchronized Java programs, particularly when using multiple processors.

5.1 Experimental framework

We performed our experiments on an IBM xSeries 445 machine running Windows 2003 Server Enterprise Edition. This machine has 16 2.2GHz Intel® Xeon® processors and 16GB of shared memory arranged across 4 boards. Each processor has 8KB of L1 cache, 512KB of L2 cache, and 2MB of L3 cache, and each board has a 64MB L4 cache shared by its 4 processors.

As benchmarks, we used two types of shared collection classes, a hashtable and an AVL tree. For each benchmark, we used several variations of synchronized and transactional implementations. For the transactional runs of each benchmark, we used three conflict detection granularities: object-level only, word-level only, and word-level for arrays.

Our baseline hashtable consists of a fixed array of 256 buckets, where each bucket is an unsorted singly-linked list of elements updated in place. This version uses coarse-grained synchronization; it synchronizes on the entire hashtable for all operations (like Java's built-in hashtable). We also implemented a fine-grained one that adds an explicit bucket object and uses that for synchroniza-

[5] Inlining allows the JIT to eliminate `I1` and `I2` altogether if it can prove that no other open for writes on the transaction record have occurred in the transaction.

tion. Additionally, we provide a transactional implementation that uses our `atomic` extension to Java and is identical to the baseline hashtable, with `synchronized` replaced by `atomic` in the source.

We also implemented a *quasi-functional* hashtable that has immutable bucket list elements. Rather than altering the old list structure in place, insertion or deletion creates a new list when necessary, much as in functional languages like ML or Lisp. Although this type of hashtable allocates additional memory, it has significant advantages with an STM: linked list traversals incur no STM overhead because existing list elements are immutable [6] and newly allocated ones are transaction-local. In fact, the quasi-functional hashtable only needs to open or log accesses to the bucket itself. While the standard hashtable requires $O(n)$ STM operations per access (where n is the average number of elements per bucket), the quasi-functional hashtable requires $O(1)$.

For our AVL tree, we have two implementations: a synchronized implementation that uses a single lock to gate all accesses to the tree and a transactional implementation that uses our `atomic` construct. For performance, an AVL tree requires periodic balancing — an operation that modifies significant portions of the tree.

Finally, to evaluate the performance of our STM system on a large benchmark, we used the OO7 object operations benchmark originally designed in the database community. We used the Java version of the benchmark as described in [38] and modified the synchronized regions to use transactions. The benchmark does a number of traversals over a synthetic database organized as a tree. Traversals either lookup (read-only) or update the database. The synchronized version allows locking at different levels of the tree, thus providing different locking granularities.

5.2 Compiler optimizations and single thread overhead

Figure 12 shows the single-threaded execution overhead of the transactional versions of the benchmarks relative to the synchronized versions. For each benchmark, we measured the execution time required to perform a random set of 1,000,000 operations using a total of 20,000 values. For the transactional runs, we used object-level contention. The `No STM Opts` bars show the STM overhead with no compiler optimizations enabled. The remaining bars show the cumulative influence of compiler optimizations: `Base STM Opts` enables standard StarJIT global optimizations for STM operations, `+Immutability` used immutable field optimization, `+Txn Local` used transaction-local object optimization, and `+Fast Path Inlining` used inlining the STM fast paths.

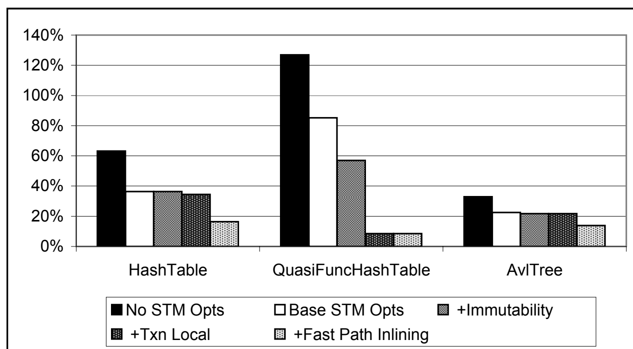

Figure 12. Single-thread execution overhead of transactional vs. synchronized programs with varying STM compiler optimizations.

The first set of bars demonstrate the overall effect of compiler optimizations on the standard hashtable. With no optimizations, the transactional code is 63% slower than the synchronized code on a single thread. Simply enabling StarJIT's standard compiler optimizations reduced this overhead to 36%. As there are few remaining immutable fields or transaction-local objects, there is little additional benefit from these optimizations. The major remaining overhead is the open for read operation (potentially) required on each list element in a bucket. Partial inlining of this read operation into compiled code reduced the final overhead substantially. With all optimizations, the atomic code is only 16% slower than the synchronized code.

The second set of bars shows the same optimizations on the quasi-functional hashtable. Here, with no optimizations, the transactional code is over a factor of two slower than the synchronized version. Our core StarJIT optimizations bring this down to 85%. Detecting and suppressing STM operations for immutable fields reduces it to 57%, and the similar optimization for transaction-local objects reduces it further to just less than 9%. As few STM open operations are left at this point, partial inlining does not give us any significant additional benefit. For this benchmark, the optimized atomic code is only 9% slower than the synchronized code.

The final set of bars shows our compiler optimizations on the balanced AVL tree. The behavior here is similar to that of the standard hashtable. In total, compiler optimizations reduced the overhead from 33% to roughly 14%.

In each of these benchmarks, the single-threaded overhead of optimized transactional code is less than 20%. For the benchmark written with an STM in mind, the overhead is less than 10%.

5.3 Scalability

Figures 13 and 14 illustrate the scalability of fully optimized transactional code with respect to synchronized code.

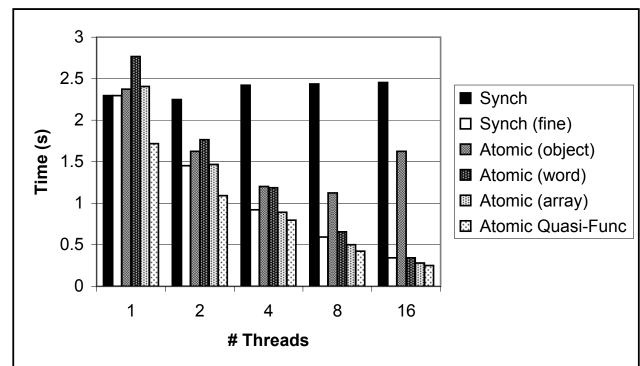

Figure 13. Hashtable execution time over multiple threads

In Figure 13, we show the performance of our hashtable implementations from 1 to 16 threads where each thread is mapped to a different processor. For each run, we used a distribution of 64,000 operations consisting of 10% insertions, 10% removals, and 80% lookups. The bars marked `Synch` represent our baseline hashtable. The bars marked `Synch (fine)` represent the fine-grained variant. The three sets of bars marked `Atomic` represent our transactional version with (1) uniform `object`-level conflict detection, (2) uniform `word`-level detection, and, (3) a mixed `array` mode using word-level detection only for array elements. Finally, the last set of bars represent the quasi-functional hashtable run atomically using word-level detection only for array elements.

On a single thread, both the coarse-grained and fine-grained synchronized implementations are equivalent in performance as they incur the same costs of acquiring and releasing a monitor. In contrast, the atomic version is about 3% slower with object-level detection and 20% slower with word-level. The overhead of

[6] This is enforced by final fields.

word-level detection has two primary causes. First, the mapping from a word to a transaction record is more complex and, thus, is more expensive to compute. Second, object-level detection uses a transaction record embedded within the object and, thus, interacts better with the memory hierarchy.

As we add threads, however, we see the benefit of both transactions and word-level conflict detection. The coarse-grained synchronized implementation does not scale at all, as all operations to the hashtable are serialized and no parallelization is permitted. On the other hand, the fine-grained implementation does scale at the cost of requiring an extra bucket object to serve as a lock. The atomic implementation scales only partially with object-level detection. In this case, the entire bucket array is the granularity of conflict. In contrast, the atomic implementation scales very well with word-level detection, matching the fine-grained implementation at 16 processors. However, the mixed array mode detection does better than both. In this case, it has the benefit of object-level detection's low overhead during the linked list traversal and the benefit of word-level detection's better scalability for the bucket array.

Finally, our quasi-functional hashtable, outperforms all of the above. Surprisingly, it even outperforms the synchronized version on a single thread. We believe that this is due to better spatial locality. Remove operations periodically reallocate each bucket list; in our GC, these allocations will be contiguous in memory. Importantly, this hashtable implementation maintains its advantage as we scale to 16 threads. It avoids the lower overhead of object-level contention management during the list traversal as the compiler is able to detect that only immutable, final fields are accessed.

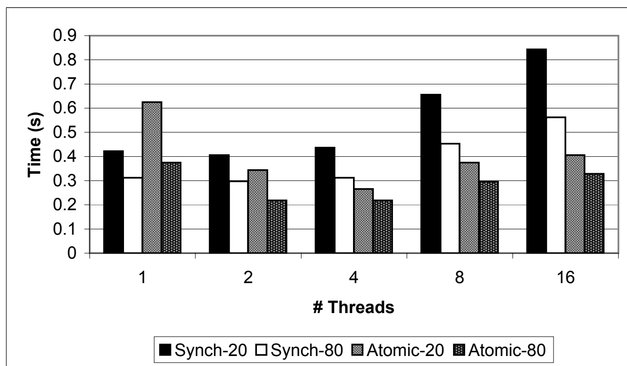

Figure 14. AVL Tree execution time over multiple threads

In Figure 14, we show the scalability of our AVL tree implementations. Here, we used two distributions of 64,000 operations. The bars marked `-80` represent executions with 80% lookups and 20% updates. The bars marked `-20` represent the reverse. As all operations are effectively gated through the root node, neither implementation scales particularly well to sixteen processors. The synchronized version requires that the root node be locked and effectively serializes all accesses. It shows no benefit at two processors and degrades beyond. The transactional version, however, scales to two processors; on an update intensive run, it scales to four processors. The transactional version provides finer-grain concurrency as it allows multiple readers to access the root simultaneously. Although the synchronized tree is faster on a single thread, the transactional version is faster on two threads and beyond.

Figure 15 studies the performance of the transactional and synchronized versions of OO7, with lookups constituting 80% of the operations. The benchmark performs 8,000 traversals equally divided between the threads. Sync L1 locks the root of the tree, and as expected, does not scale. Sync L3 and L6 lock at lower levels

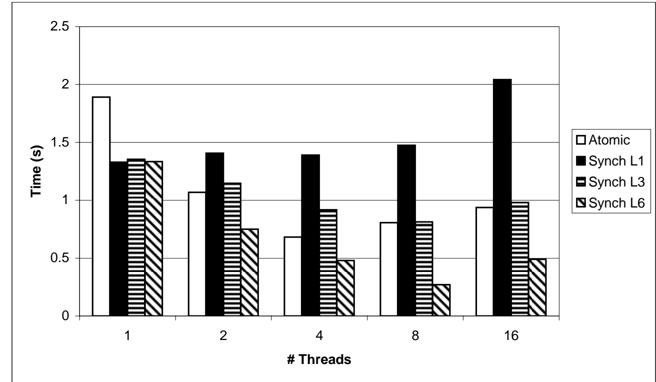

Figure 15. OO7 execution time over multiple threads

of the tree and show better scalability. The atomic version treats the entire traversal as a transaction. In effect, the granularity of the atomic version is the same as Sync L1. As shown in the figure, the atomic version performs much better than Sync L1 and approaches the performance of finer-grained locking (Sync L3 and L6).

5.4 Discussion

Our quasi-functional hashtable highlights some significant advantages of STMs and of our implementation in particular. This hashtable is similar to Java 5's ConcurrentHashMap,[7] as both use lists with immutable structure. However, ConcurrentHashMap is much more complex as it relies on clever use of standard Java synchronization for scalability. The main performance techniques used in ConcurrentHashTable at the source level are automatically provided by our system. First, to enable fine-grain concurrency, ConcurrentHashMap maintains a separate array of Java lock objects where each protects a set of slots in the bucket array. In contrast, our system automatically provides word-level contention management to get very similar fine-grain concurrency without user effort. Second, on a get operation, ConcurrentHashMap does not require synchronizing the linear traversal through the linked elements. Instead, it explicitly validates the result to ensure consistency. If validation fails, it repeats the traversal. Again, our system provides this automatically, with no user effort, through read versioning. As a get operation performs no writes, our implementation will acquire no locks. Instead, any read to mutable memory is validated at commit. Third, our JIT is able to detect that list links are final (and thus immutable). This obviates the need to log link traversal operations and results in fast constant-time validation. Finally, our system automatically inserts the control flow and bookkeeping to repeat the get operation if necessary. Incorporating these features into ConcurrentHashMap was important for performance but added significant effort, code, and complexity. In contrast, our transactional, quasi-functional implementation is no more complex than our standard, coarse-grained hashtable. The only significant difference is that the bucket lists are written in a functional ML/Lisp style rather than a typical C/Java style.

6. Related work

In lieu of locks, the recent High-Productivity Computing Systems languages — Fortress[2], X10[9] and Chapel[11] — all specify an `atomic` statement for concurrency control. Although details on their implementation are not yet available, these languages demon-

[7] ORP and StarJIT do not yet support Java 5 generics or classfiles, so we cannot do a direct comparison at this time.

strate the trend towards using transactions as the concurrency control mechanism in emerging languages.

Several researchers have implemented TM language constructs. Harris and Fraser [16] add to the Java language a conditional critical region (CCR) statement of the form `atomic(E){S}`, which atomically executes `S` when the expression `E` evaluates to true. We adopt their techniques for cloning methods in the JVM and dealing with native method calls, and we build on their technique for detecting conflicts at the memory word granularity.

Harris, Marlow, Peyton-Jones, and Herlihy [17] add a transactions monad to Concurrent Haskell and introduce *composable memory transactions* with two functions to block and compose nested transactions: (1) `retry`, which blocks a transaction until an alternate execution path becomes possible, and (2) `orElse`, which composes two possibly-blocking transactions as alternatives. We build on [17] by adding `retry` and `orelse` operations to an imperative, object-oriented language (Java) running on a multiprocessor.

AtomCAML [33] extends Objective Caml with a new primitive synchronization function (`atomic`) of type `(unit->'a)->'a` that executes its function argument atomically. The AtomCAML STM implementation[33] updates memory in place and uses an undo log to roll back side effects. Their implementation runs on a uniprocessor and uses scheduler-based techniques to provide atomicity. Shinnar, Tarditi, Plesko, and Steensgaard [36] use undo logging on a uniprocessor to implement a new language construct that restores memory state on an exception.

Shavit and Touitou [35] introduce the term STM and present a *static* STM, which requires advance knowledge of the memory locations involved in a transaction. Herlihy, Luchangco, Moir, and Scherer's DSTM [18] and Marathe, Scherer, and Scott's ASTM [24] provide an *object-based* STM API for Java, while Fraser's FSTM [13] provides an *object-based* STM API for C. Marathe, Scherer and Scott [25] compare the performance and design tradeoffs of DSTM and FSTM. Object-based STMs create a writable clone of an object on an open for write. On a commit, the transaction atomically makes the clone the valid version visible to other transactions. To handle read-after-write dependencies, an open for read on an object O checks whether the transaction has already created a writable clone of O. Object cloning requires the programmer to wrap a transactional object with a new type — for example, `TMObject` in DSTM and `stm_obj` in FSTM — that indirectly references the object. All references to a transactional object must go through its wrapper. Marathe et. al.'s measurements [25] show that going through extra levels of indirection (DSTM has two levels of indirection, while FSTM has one) hurts performance. Cloning is also expensive for large objects such as arrays. Ananian and Rinard suggest handling large arrays using functional arrays[4]. Because object cloning requires the programmer to use wrappers, it precludes the programmer from using existing library code transactionally. Welc, et. al., [37] propose transactional monitors, which implement Java monitors as lightweight transactions, but they do not consider compiler optimizations.

The STM implementations of [16] and [17] buffer memory updates in a per transaction write buffer. On a commit, the transaction updates the written memory locations from the write buffer after acquiring ownership of those locations — Marathe et. al. [25] refer to this as *lazy acquire* and show how this affects performance. Write buffering makes individual accesses expensive: to handle read-after-write dependencies, a read of a memory location L first checks the write buffer data structure in case the transaction has previously written to L.

Past multiprocessor STM implementations provide non-blocking property guarantees. Ennals [12] presents a lock-based STM and argues that STMs do not need to be non-blocking. Our TM system builds on the McRT-STM [34], which uses two-phase locking. Preemption safety is a desirable property, but it should become less of a concern as future processors use Moore's law to increase the number of processing cores. In the long run, memory will be but one transactional resource in a potentially distributed environment, and TM would need to be integrated with a general transaction monitor — a lock-based STM is necessary in such a setting.

Hardware transactional memory (HTM) leverages existing cache coherence logic to implement transactions efficiently in the cache [19, 31]. To support a general purpose transactional memory language construct, HTMs must support transactions of arbitrary footprint and duration either purely in hardware or in combination with software. Virtualized Transactional Memory [32] and Unbounded Transactional Memory [3] support transactions of unbounded footprint and duration. Large Transactional Memory [3] supports transactions whose footprint fits within physical memory. Log-based Transactional Memory (LogTM) [28] supports transactions of unbounded footprint. Similar to our approach, LogTM updates memory locations in place and uses an undo log to roll back side effects.

Hybrid transactional memory [27, 4, 21] combines HTM with STM: a transaction first executes using HTM and then falls back on STM if the HTM aborts. The techniques we describe should integrate nicely with HTM as a hybrid solution.

Carlstrom et. al. [8] map the Java concurrency features onto the TCC HTM system [15] and discuss mismatches between existing Java features and transactional execution, most notably the condition synchronization methods `wait`, `notify` and `notifyAll`.

7. Conclusions

Transactional language constructs in a modern imperative language such as Java not only provide a powerful concurrency control mechanism but also enable JIT compiler and runtime optimizations that target their overheads. This paper is the first to address JIT compiler optimizations for transactional memory. We extended Java with new syntax that provides composable nested transactions, and we implemented a high-performance software transactional memory system integrated into a managed runtime environment. We demonstrated how a tight integration between an optimizing dynamic compiler and a well-tuned runtime results in a high performance software transactional memory system. We described new compiler optimizations that specifically target STM overheads and showed how to model STM operations in a compiler intermediate representation so that existing global optimizations target STM overheads correctly, especially in the presence of nested transactions. We also showed how supporting both object level and word level conflict detection on a per-type basis improves scalability while minimizing overhead.

The results on a 16-way SMP system demonstrate that using our techniques, performance of transactional memory in Java can compete with that of well-tuned synchronization, especially as the number of processors increases. On a single processor, StarJIT's optimizations reduced STM overhead for the hashtable and AVL tree benchmarks to 16% or less compared to their lock-based versions. The STM overhead was even less (under 10%) for the quasi-functional hashtable. On multiple processors, a combination of object- and word-granularity conflict detection achieved the best overall scalability.

Acknowledgments

We'd like to thank Dan Grossman and the anonymous reviewers for their feedback on this paper. We'd also like to thank Rick Hudson, Leaf Petersen, Jim Stichnoth, Jesse Fang, and other members of the Programming Systems Lab for the many discussions on transactional memory.

References

[1] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. S. Menon, B. R. Murphy, M. Serrano, and T. Shpeisman. The StarJIT compiler: a dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1), February 2003.

[2] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification version 0.707. http://research.sun.com/projects/plrg/fortress0707.pdf, 2005.

[3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA 2005: High-Performance Computer Architecture*.

[4] C. S. Ananian and M. Rinard. Efficient object-based software transactions. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*.

[5] A. W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2), 1989.

[6] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. Jun 2005.

[7] P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Software—Practice and Experience*, 27(6), June 1996.

[8] B. D. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. C. Minh, L. Hammond, C. Kozyrakis, and K. Olukotun. Transactional execution of Java programs. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*.

[9] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarakar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005: Object-Oriented Programing, Systems, Languages, and Applications*.

[10] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Intel Technology Journal*, 7(1), February 2003.

[11] Cray Inc. Chapel Specification 0.4. http://chapel.cs.washington.edu/specification.pdf, 2005.

[12] R. Ennals. Software transactional memory should not be obstruction-free. http://www.cambridge.intel-research.net/˜rennals/notlockfree.pdf, 2005.

[13] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Technical Report UCAM-CL-TR-579.

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

[15] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *ASPLOS-XI: Architectural Support for Programming Languages and Operating Systems*, 2004.

[16] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA 2003: Object-Oriented Programing, Systems, Languages, and Applications*.

[17] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP 2005: Principles and Practice of Parallel Programming*.

[18] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC 2003: Principles of Distributed Computing*.

[19] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA 1993: International Symposium on Computer Architecture*.

[20] W. N. S. III and M. L. Scott. Contention management in dynamic software transactional memory. In *PODC 2004 Workshop on Concurrency and Synchronization in Java programs*.

[21] S. Kumar, M. Chu, C. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *PPoPP 2006: Principles and Practice of Parallel Programming*.

[22] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2), 1981.

[23] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL 2005: Principles of Programming Languages*.

[24] V. Marathe, W. Scherer, and M. Scott. Adaptive software transactional memory. Technical Report Technical Report 868, University of Rochester, 2005.

[25] V. J. Marathe, W. N. Scherer, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *LCR 2004: Languages, Compilers, and Run-time Support for Scalable Systems*.

[26] V. S. Menon, N. Glew, B. R. Murphy, A. McCreight, T. Shpeisman, A.-R. Adl-Tabatabai, and L. Petersen. A verifiable SSA program representation for aggressive compiler optimization. In *POPL 2006: Principles of Programming Languages*.

[27] M. Moir. Hybrid hardware/software transactional memory. http://www.cs.wisc.edu/ rajwar/tm-workshop/TALKS/moir.pdf, 2005.

[28] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *HPCA 2006: High-Performance Computer Architecture*.

[29] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and preliminary architecture sketches. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*.

[30] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java. In *CC 2005: International Conference on Compiler Construction*, Lecture Notes in Computer Science 2622.

[31] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO 34: International Symposium on Microarchitecture*, 2001.

[32] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA 2005: International Symposium on Computer Architecture*.

[33] M. F. Ringenburg and D. Grossman. AtomCaml: first-class atomicity via rollback. In *ICFP 2005: International Conference on Functional Programming*.

[34] B. Saha, A.-R. Adl-Tabatabai, R. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP 2006: Principles and Practice of Parallel Programming*.

[35] N. Shavit and D. Touitou. Software transactional memory. In *PODC 1995: Principles of Distributed Computing*.

[36] A. Shinnar, D. Tarditi, M. Plesko, and B. Steensgaard. Integrating support for undo with exception handling. Technical Report MSR-TR-2004-140, Microsoft Research, December 2004.

[37] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *ECOOP 2004: European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*. Springer-Verlag.

[38] A. Welc, S. Jagannathan, and A. L. Hosking. Revocation techniques for Java concurrency. In *Concurrency and Computation:Practice and Experience*. John Wiley and Sons, 2005.