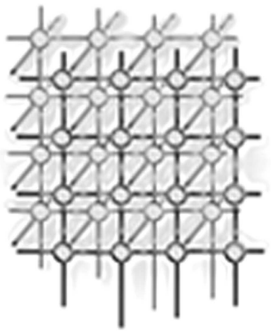
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2008; **00**:1–7



Compiler and Runtime Techniques for Software Transactional Memory Optimization

Peng Wu*, Maged M. Michael, Christoph von Praun, Takuya Nakaike, Rajesh Bordawekar, Harold W. Cain, Calin Cascaval, Siddhartha Chatterjee, Stefanie Chiras, Rui Hou, Mark Mergen, Xiaowei Shen, Michael F. Spear†, Hua Yong Wang, Kun Wang

IBM Research
†*University of Rochester*

SUMMARY

Software transactional memory (STM) systems are an attractive environment to evaluate optimistic concurrency. We describe our experience of supporting and optimizing an STM system at both the managed runtime and compiler levels. We describe the design policies of our STM system, and the statistics collected by the runtime to identify performance bottlenecks and guide tuning decisions. We present initial work on supporting automatic instrumentation of STM primitives for C/C++ and Java programs in the IBM XL compiler and J9 JVM. We evaluate and discuss the performance of several transactional programs running on our system.

KEY WORDS: transactional memory, software transactional memory, compiler optimization, optimistic concurrency

1. Introduction

It is increasingly clear that transactional memory (TM) will be supported in some form by most of the major systems manufacturers in the near future. Examples include recent announcements for hardware support from Sun Microsystems [32], the release of a transactional compiler and runtime system by Intel [8], and Azul systems using a transaction-like form of speculative lock

*Correspondence to: IBM T.J. Watson Research Center, P.O.Box 218, Yorktown Heights, NY, USA

elision [31]. However, despite over fifteen years of research since the conception of transactional memory by Herlihy and Moss [17], there has been limited work exploring the interaction of TM with compilers and managed runtimes [1, 14, 29]. Most research has instead focused on the construction of scalable, low-overhead, transactional memory systems, some purely in software [16, 14, 20, 28], some with significant hardware support [2, 6, 12, 22, 24], and some in-between [4, 9, 27, 36], while largely ignoring the potential enablers of static analysis and dynamic optimization as a means of reducing the overhead of transactional execution.

In this paper, we describe our early experience building and optimizing a complete system stack around an all-software transactional runtime system, with an emphasis on application-driven optimization of TM bottlenecks. We analyzed several applications with the transactional runtime system and we identified many opportunities for reducing transactional overheads. To take advantage of these opportunities, we implemented support to automatically instrument shared references to STM primitives for C/C++ and Java programs in IBM's XL optimizing compiler and J9 Java virtual machine.

The initial focus of our compilation work is the overhead reduction of unnecessary read and write barriers for operations that are known to be conflict-free. We find significant opportunity for reducing barriers to address-taken stack addresses. We also present results on escape analysis for eliminating heavyweight read and write barriers on heap memory that is not shared by other threads at the point of reference. And we discuss the need and issues with memory checkpointing that were largely left to be dealt with manually by prior work. We discuss these optimizations in detail in Section 3 (for C/C++) and in Section 4 (for Java).

In addition to describing our compiler and JVM implementation, we also include an evaluation of three benchmarks: SSCA2, HSQLDB, and a B+tree algorithm based on the Aries database system. We include the general analysis of their scaling properties and characteristics of their transaction working sets, in addition to optimization studies using our compiler and JVM. This performance analysis is presented in Section 5.

2. Software Transactional Memory Implementation

We begin with a description of the STM system used in our study. We describe the statistics collected by the STM runtime and how they can be used to provide feedback to components of the transactional memory software stack (e.g., application, compiler, managed runtime), and to improve the STM system itself.

2.1. STM Implementation Features

In designing our STM, we selected policies and mechanisms that favor enabling concurrency, avoiding restrictions that limit the use of TM, and minimizing the performance overheads.

We use a block-based (8-byte) mapping of shared memory locations to shared STM metadata that controls access and detects conflicts [13, 26, 10]. This has the advantage of being applicable to any programming languages, unlike object-based mapping which is mostly suitable to object-oriented programming languages.

For storing data values, our implementation uses a buffered-writes policy [13, 21, 10]. The values to be written are buffered in transaction-private structures and written to shared memory only after the transaction is guaranteed to commit. This policy simplifies memory management, and prevents other threads from observing values that may never be committed, enforcing the isolation properties of TM.

It also follows a policy of invisible reads [26, 21, 10]. Specifically, reads in the course of a transaction do not result in any writes to shared metadata. This has the advantage of enabling concurrent reads of the same location to proceed without inter-transaction interference that otherwise may result from cache coherence traffic if visible writes were employed.

As a general rule, the STM implementation uses a strategy of disjoint access parallelism (except for hashing collisions in block-based mapping). In other words, concurrent non-conflicting transactions (i.e., transactions that do not write to a block that has been accessed by another transaction) should all commit successfully. Furthermore, such transactions should not interfere with another transaction's progress. In addition to motivating our choice of invisible reads, this also led to our choice of read-set validation mechanism [16, 21] and the avoidance of global timestamps [10, 30, 25]. The latter requires writes to shared metadata by non-conflicting transactions.

To simplify memory management, in particular to avoid restrictions on the memory reclamation of objects accessed in transactions, we do not require the STM operations to be non-blocking, but rather use locking in metadata operations [11, 26, 10].

2.2. STM Runtime Statistics

In order to characterize applications and identify performance bottlenecks, we implemented a rich set of statistics that are collected by our STM runtime. These statistics have provided insights into application behavior and have driven the compiler optimizations discussed later in the paper. An overview of these statistics is presented below.

Read-/write-sets: The size of read-/write-sets are inherent characteristics of the application code. They are affected by the effectiveness of barrier instrumentation either by the programmers or the compiler. In our compiler tuning, we use these statistics to evaluate the effectiveness of compiler optimizations to eliminate unnecessary barriers.

Duplicate reads and writes: arise when there are multiple read- or write-barriers to the same location in the same transaction. A high percentage of duplicate accesses indicates the potential to compact the read-/write-set to reduce the overhead of read-barrier and read-set validation.

Access granularity: A high percentage of reads and writes that match the same conflict unit as prior reads and writes indicates potential optimizations in the compiler and the STM runtime to use lightweight versions of read and write barriers without unnecessary conflict detection operations which are already covered by the earlier accesses in the transaction that matched the same conflict sets.

Read after subset of writes: A high percentage of reads that follow some write in the same transaction indicates the importance of employing fast lookup mechanisms in the STM design such as Bloom filters [5]. On the other hand, a low percentage indicates that the

overheads of managing the Bloom filters are unnecessary. Thus the STM implementation can be reconfigured to match the applications characteristics.

Reads after writes: A high percentage of reads that are of locations previously written by the same transaction indicates that when allowable it may be beneficial for the STM design to use an in-place write policy, or alternatively employ record-keeping mechanisms that allow fast access to buffered to-be-written values.

Read-only transactions: If read-only transactions are more common than read-write transactions then it may be possible to employ global timestamp mechanisms [10] that may otherwise limit scalability under heavy loads of read-write transactions, but scale well under read-only transactions.

Silent writes: If the percentage of silent writes is high then it may be beneficial to transform silent writes into reads. This can reduce the size of the write-set that decrease the write-set search time for the read-barriers, and reduce the number of expensive atomic instructions per transaction to register the write.

Retries: A large number of retries can indicate either that the application lacks inherent concurrency, or that certain STM design choices are causing unnecessary conflicts. Differentiating between these cases can be done by collecting these numbers with different STM configurations. If an application doesn't scale despite a low retry rate, then this indicates that the culprit is likely not related to synchronization but rather other causes such as data layout and cache performance. Identifying the causes of retries can provide input to the STM regarding retry policies. It can also provide input to the programmer on application data layouts that may cause synchronization conflicts.

All these have been useful in optimization. In the next section we discuss compiler optimizations that reduce the number of read/write barriers and how this affects performance.

3. C/C++ Compiler Instrumentation for STM

The major obstacle of programming an STM is the need to explicitly instrument all potentially conflicting memory references with read-/write-barriers. To improve the usability of the STM system, we implemented an instrumentation pass in a development version of the IBM XL compiler. This pass instruments all potentially conflicting references in transactional scopes marked in the source code to STM read-/write-barrier calls.

In this section, we discuss the compilation issues we faced when instrumenting C/C++ programs for STM, and describe several compiler and STM optimizations to reduce overheads in instrumented codes.

3.1. Memory checkpointing

When a transaction writes to a stack location or a privately allocated memory that has not yet escaped the thread (i.e., a memory location that is not yet visible to other threads), the write does not induce any conflict. We refer to such writes as *contention-free writes*. Such writes do not require write-barriers, but may need to be checkpointed if the overwritten values need to be recovered upon a retry.

```
#pragma omp parallel private(s,count)
{ ...
   s = (int *) malloc(n*sizeof(int));
   for (...) {
     ...
     TM_BEGIN {
      ...
      if (...) s[count++] = w;
      ...
     } TM_END
   }
   ... = s[...];
}
```

Figure 1. A checkpoint example from ssca2.

```
void foo(node_t **leaf, int *offset){
   ...;
   *leaf = ...;
   *offset = ...;
}

node_t *leaf = NULL;
int offset = -1;
TM_BEGIN {
   foo(&leaf, &offset);
   if (!leaf && offset != -1)
   ....
} TM_END
```

Figure 2. An address-taken example from b+tree.

Figure 1 shows a code extracted from ssca2 (details in Section 5.2, where transactions are marked by `TM_BEGIN` and `TM_END` macros. In this example, `s` is declared OMP private and points to a chunk of memory allocated within the thread, which remains private until the end of the transaction; and `count` is a private integer variable. Therefore, the writes in `s[count++]` in the transaction do not cause any conflict. However, since `s[]` is live both upon entry and exit of the transaction block (due to the transaction being inside a loop), its writes need to be checkpointed. Similarly, `count` requires checkpointing as it has an exposed use inside the transaction (e.g., `count++`).

Using data-flow analysis, the compiler can exclude from memory checkpointing writes to most contention-free locations. Basically, if a variable or heap location is private to a transactional lexical scope (e.g., transactional block or procedure), that is, the variable is not live upon entry and exit to the lexical scope, then the write requires no checkpointing.

However, it is often the case that contention-free writes do have uses after exiting the transaction. In this case, checkpointing can still be avoided if the location is not live upon entry to the transactional scope, and if the write to the location dominates transaction end. The latter guarantees that, upon retry, the location will always be re-defined thus no need to recover the original value.

Finally, uninitialized locations upon a transaction entry require no checkpointing. This also includes the case when a heap location is allocated within the transaction.

3.2. Handling address-taken stack variables

In C and C++ code, programmers may have pointers to stack locations by taking the address of stack variables. The compiler will conservatively instrument writes through these pointers as write-barriers, assuming that they may point to shared memory locations. Consider the

example in Figure 2 from `b+tree` (details in Section 5.1. When instrumenting function `foo`, the compiler may assume that pointers `leaf` and `offset` refer to potentially shared memory locations, and instrument `*leaf=` and `*offset=` as write-barriers.

There are two issues when treating stack addresses the same as other addresses in a write-barrier:

- **Out of scope writes**: Our STM buffers writes in an internal write-list, which are not committed until the transaction ends. If the write buffer contains a stack address from a procedure invoked by the transaction, and at the commit time, the procedure stack is already reclaimed, then the delayed write to the memory location may corrupt stacks of other procedures.
- **Aliasing writes**: If writes to the stack can be buffered, subsequent references to the same location must also be instrumented in order to first look it up from the write buffer. This means that any direct reference of address-taken stack variables within the transaction also need to be instrumented into barriers. For instance, in Figure 2, references to stack variables `leaf` and `offset` in the callee function need to be instrumented in order to read buffered values.

To address both issues, the STM provides special filtering of stack addresses in the write-barrier so that they are neither buffered nor checked for conflicts, based on a list of compiler generated address ranges. The compiler generates this list, at the beginning of each procedure, for all address-taken stack variables that are referenced in transactions. Note that, for stack addresses that are taken outside the transaction but referenced inside, it must be ensured that the addresses are not passed to a shared pointer. In our prototype implementation, we assume that is the case.

With the filtering technique, address-taken stack addresses are never buffered, nor would they cause any conflicts, thus no barriers are necessary for any direct reference of address-taken stack variables in the transaction.

3.3. Optimizing for non-escaping heap locations

If a memory location is allocated within a thread and has not escaped the thread at the point of reference within a transaction, the reference to it does not cause any conflicts, and may not require read-/write-barriers.

However, not all references to contention-free locations can avoid read-/write-barriers. Consider a read access to a contention-free location, the read-barrier can be safely removed only if the location may not be referenced by any prior write-barriers. The issue is similar to the one discussed above on taking the address of stack variables. Otherwise, the read must be instrumented as a read-barrier, but a lightweight one that can skip the book-keeping for conflict detection. Consider a write access to a contention-free location, the write-barrier can be safely replaced by memory checkpointing, only if the location may not be referenced by any prior write-barriers. The checkpointing may be eliminated if it further satisfies the conditions specified in Section 3.1.

4. Java Runtime and Compiler Support for STM

4.1. Instrumentation Issues

During Java byte-code or jitted code execution, three types memory may be accessed: Java stack, Java heap (e.g., for field or array accesses), or JVM metadata (e.g., class information).

Since the Java stack is always thread-private, stack access is excluded from instrumentation. Since the heap is shared among threads, heap access needs to be instrumented by the interpreter when it executes the byte-code, and by the JIT-compiler in the jitted code. In the prototype implementation, we modified the JIT-compiler to replace every field and array access with read- and write-barriers to the STM library. The interpreter is not modified. Instead, every method is compiled at its first invocation (i.e., only jitted code is executed). Field or array accesses could also occur in native code. We found two such cases that require read- and write-barrier instrumentation: class loading and array copy. In general, java.lang.System.arraycopy is implemented in native libraries using the memcpy function. In both cases, we added STM read- and write-barriers in the native implementation of those routines.

Note that JVM metadata may be subject to concurrent access as well. However, we found it impractical to instrument these accesses. First, compiler instrumentation may not be feasible since some JVM code is written in assembly, and the sheer size of the JVM prevents manual instrumentation. Secondly, if we were to instrument all JVM metadata, accesses to the jitted code itself would require instrumentation too as JIT-compilation may occur during transaction. This would significantly degrade the JVM performance. Hence we decided not to instrument JVM metadata. Instead, for the purposes of our early prototype, we treat such accesses to meta-data as non-transactional, and avoid conflicts by forcing compilation and mutating shared metadata accesses to occur before program execution.

4.2. Optimization Issues

We implemented two optimizations: versioning of transactional calls and barrier elimination of transaction local objects. We briefly explain how these optimizations are implemented.

- When a method is invoked, it is compiled as a non-transactional version without inserting read-/write-barriers. At this time, we add a special prologue to the entry of the compiled code. In the prologue, we check whether or not a transaction is active. If a transaction is active, we call the compiler to generate a transactional version, and then patch the branch instruction to call this transactional version. For indirect call-sites, i.e., method calls through the virtual function table, it is possible to add entries for transactional methods in the virtual function table. For our benchmark this was not yet needed and therefore we have not yet implemented this optimization.
- Some field accesses can be exempted from instrumentation. For example, field access to an object which is allocated inside the transaction, we call those objects transaction-local. For example, we can safely eliminate read-/write-barriers in the constructor, which is called immediately after an object is created. This optimization is similar to the one described in Section 3.3.

5. Application Characterization and Evaluation

We chose three workloads with different characteristics to evaluate the STM implementation and related compiler techniques.

B+tree is a kernel program in C that exercises concurrent accesses to a B+tree structure. The program can be regarded as classic case for transactional memory, since thread-safety of the sequential B+tree data structure is achieved mostly by encapsulating tree operation into coarse-grained transactions. ssca2 implements a parallel graph analysis algorithm with irregular memory access and very fine granular transactions. hsqldb is a Java in-memory database that we use to evaluate Java-specific issues. Similar to b+tree, it presents the case of a simple parallelization of an otherwise sequential database implementation: A coarse-grained transaction is used per SQL statement to achieve serializability of concurrent accesses to the database.

The STM implementation enables concurrency inherent in the applications but it cannot eliminate inherent scalability bottlenecks. Therefore, the performance of applications with inherent concurrency is expected to scale well when using STM. The main challenge for STM implementations is reducing their high contention-free overhead over sequential performance. The compiler techniques that we explore focus primarily on reducing these overheads.

5.1. B+Tree

The btree is the core data structure used for building database indexes to store key-value pairs. In a btree of order m, every non-root node can store at least $m-1$ and at most $2m-1$ keys. The keys are stored in non-decreasing order. We have implemented a version of the btree called the B+tree in which nodes at the same level are connected via a singly-linked list [19]. In our implementation, only the leaves store the values. Each leaf has the same depth, which is the tree's height, h. For a B+tree of order m containing n keys, the height h is $O(log_m(n))$.

B+tree can support three types of operations: insert a key-value pair, delete a key-value pair, and fetch the value of a given key. Both insert and delete operations change the number of keys stored in the tree and can lead to structural re-balancing to preserve B+tree's structural constraints. Each B+tree operation starts at the tree root and descends down to the leaves. Any leaf update can trigger a structural modification to rebalance the tree. The structural modification can potentially affect the entire tree.

In our evaluation, the lock-based implementation treats tree operations as critical sections and serializes them using coarse-grained locks over the entire tree. In the transactional implementation, the critical sections are replaced by transactions.

Figure 3 shows the performance of the lock-based and STM versions for a workload of inserting 64K items into the tree. The critical section accounts for 94% of the overall execution time of this workload. The average critical section size amounts to 629 dynamic instructions for this workload.

We first observe a high contention-free overhead of using STM (10 times over the lock-based version). This is primarily due to the large number of dynamic read-barriers (110 on average) that inflate the dynamic instruction counts of the STM version. Detailed analysis of this overhead is presented in Section 5.1.1. However, as the number of threads increases, the

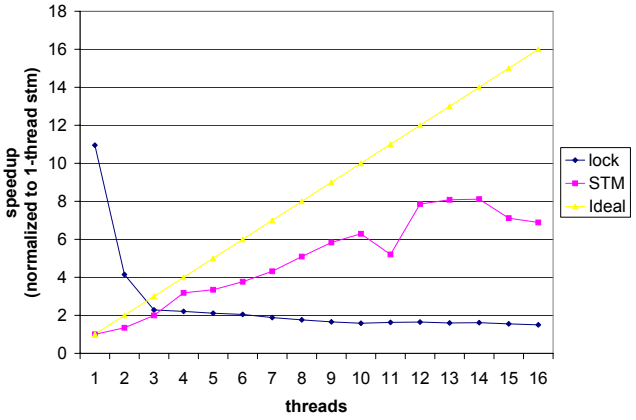


Figure 3. Scalability of B+tree on a 16-way Power5.

STM implementation can at runtime detect and exploit avenues of parallelism, resulting in reasonable scaleup.

On the other hand, due to the coarse-grained nature of the lock, performance of the lock-based version degrades as the number of threads is increased. The lock-based performance can be improved by changing the lock granularity. However, implementing a correct and efficient fine-grained lock-based concurrent algorithm is very cumbersome.

5.1.1. Compiler evaluation

The application uses very coarse-grained transactions to guard the entire operation of fetch, insert, and delete. There are 3 transaction blocks and 13 procedures called from the transactions that need to be instrumented. The compiler instrumented over 100 static write-barriers, and over 200 static read-barriers.[†]

For a single thread run of 64K insertions, the average and maximum read-set sizes reported by STM runtime are 110 and 310 entries, respectively. The average number of dynamic read-barriers are 30% higher than the read-set size due to duplicate reads. The average and maximum write-set size are 5.67 and 278 entries, respectively. The wide range of write-set size is due to the occasional re-balancing of the tree that may modify many more nodes than the usual insertion.

Figure 4 shows the percentage of performance improvement due to compiler optimizations. The handling for address-taken stack variables improves the performance, on average, by 26%. For a single thread run, 20% of the dynamic write barriers are to address-taken stack addresses.

[†]The number of barriers instrumented by the compiler do not map exactly back to source code instrumentation due to compiler code replication prior to the instrumentation pass.

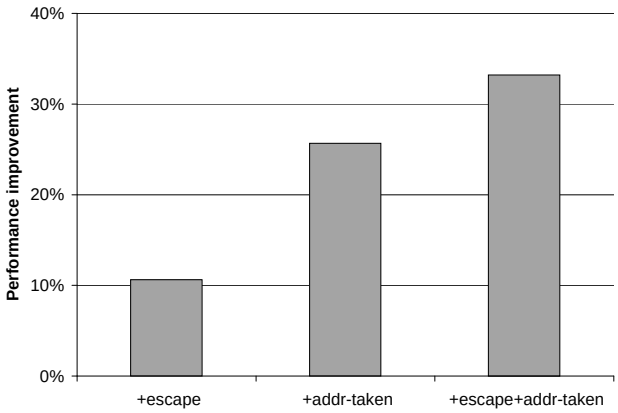


Figure 4. Performance improvements from compiler optimization for B+tree with 64K insertion operations, amortized over speedups of 16 runs, ranging from 1 to 16 threads.

Using the technique described in Section 3.2, these write-barriers become very lightweight and involve only a few compares. It also reduces 2% of the read-barriers. More importantly, since stack write-barriers do not add to the write-list, the optimization effectively reduces the write-list size by 27% for a single-thread run. Note that, since write-barriers accounts for less than 5% of total number of barriers, we believe that the performance gain comes from the reduction of write-list size, which reduces the search time for each read-barrier.

Barrier elimination for non-escape heap locations accounts for 10% improvement, on average. The opportunity lies in function `create_bptnode`, which is invoked for each insertion to allocate and initialize a new node. In `create_bptnode`, the malloced location does not escape the function. Our escape analysis is able to eliminate all but one of the write-barriers in this function. For a single thread run, the optimization reduces the number of dynamic write-barriers by 40% and the write-list size by 30%. We also notice that there are additional initialization of the newly allocated tree node following the invocation of `create_bptnode` and through the pointer returned. The write-barriers for some of the initializations may be safely removed using inter-procedural escape analysis.

Similar to stack address filtering, the performance gain most likely comes from the reduction of write-list size. Note that the improvement from escape analysis is less than that from the previous stack optimization, even though it has a larger reduction of the write-list. We speculate that the writes reduced by stack filtering happen to occur earlier in this workload, thus its impact to the search time of read-barriers is more significant.

critical section (file:line Number)	*coverage phase [%]*	*count*	*size [# Insts]*	*read-set*		*write-set*	
				avg	*total*	*avg*	*total*
betweennessCentrality.c:131	2.14	46674	10	2.0	434	1.0	217
betweennessCentrality.c:306	30.80	376962	28	3.1	3604	1.1	3570
betweennessCentrality.c:380	5.28	107419	17	4.0	464	1.0	219

Table I. Transaction characteristics in ssca2.

5.2. SSCA2

Table I shows the dynamic characteristics of the critical sections in ssca2 [3]. The *coverage* metric refers to the fraction of the execution spent in a particular critical section. The *count* metric refers to the number of transactions that successfully commit, and *size* specifies the number of instructions in the critical section without STM instrumentation. The number has been determined using the lock-based code, starting to count at the instruction after the lock is acquired and stop the count just before the lock is released. *read-set* and *write-set* report the *average* number of unique word-aligned non-stack locations that are read or written within a transaction and the total number of unique word-aligned non-stack locations across all transactions; the same location may fall into the read and write set if it is first read, then written.

We focus the evaluation on kernel 4 of ssca2 that computes the "betweenness centrality" metric for each node in a randomly generated, clustered graph; graph generation is done in another kernel of the benchmark. Two variants of the algorithm are implemented that execute in two different phases. The critical section at line 131 falls into one phase while the critical sections lines 306, 380 are executed in the other phase. For both variants of the kernel, a significant fraction of the execution is spent outside critical sections, hence not contributing to task-interdependence.

Figure 5 illustrates the execution times of ssca2 when executed on up to 16 threads, choosing input scale $k = 13$ (approximately $10K$ nodes and $80K$ edges in the graph). For high performance, this benchmark relies on efficient data access in cache. We observed that in configurations with two or more threads, threads execute on cores that do not share the same L2 cache on our architecture, resulting in a significant increase of L2 misses and L2 interventions. Consequently the CPI metric increases from 1.4 to 2.1 to 2.6 when moving from one to two to four threads.[‡] Hence the reason for the lack of scalability documented in the chart is the increase in data access latency, not the synchronization. Even at 16 threads the transaction rollback rate is very low (2.5%). For 12 threads and beyond, the partitioning of the work among threads is unbalanced; an artifact that we attribute to the relatively small data set size.

[‡]CPI is measured on the basis of the fine-grain lock implementation. STM operations significantly deflate CPI numbers since operations tend to have good cache locality.

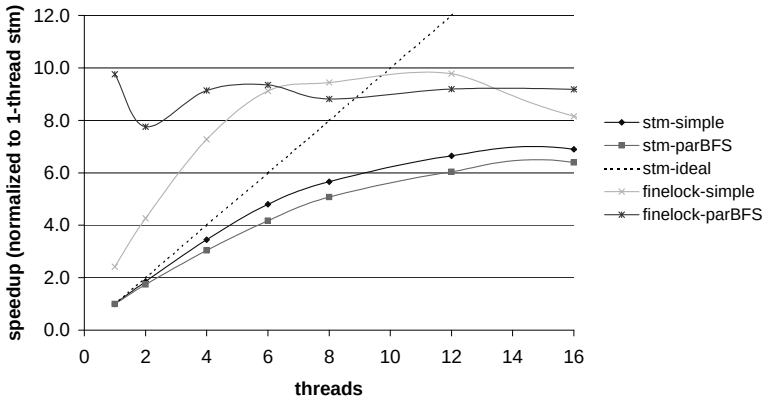


Figure 5. Scalability of ssca2 on a 16-way Power5.

Program	**read-barrier** manual (compiler)	**write-barrier** manual (compiler)	**checkpoint** manual (compiler)
ssca2	3.3 (3.69)	1.01 (0.93)	0.07 (0.15)

Table II. Dynamic barrier counts of ssca2 per transaction instance with manual and compiler instrumentation

5.2.1. Compiler evaluation

ssca2 contains four independent transaction blocks and a function that need to be instrumented. For this application, we are able to compare compiler instrumentation with manual instrumentation.

Overall, compiler barrier instrumentation matches well with manual instrumentation. As shown in Table II, the compiler generates 7% fewer dynamic read-barriers and 8% more dynamic write-barriers. In kernel parBFS, the compiler is able to remove an array reference via common subexpression elimination prior to the instrumentation, thus instrumented one less static read-barrier than manual instrumentation. On the other hand, one additional write-barrier and read-barrier are instrumented by the compiler for references to non-escaping heap locations. In both cases, the allocation of the heap and the reference to the heap location occur in separate functions after the compiler outlined OMP parallel regions. Our intra-procedural escape analysis is not able to eliminate these barriers. Therefore, the compiler version generated fewer checkpoints but more write-barriers.

5.3. HSQLDB

HSQLDB [18] is a SQL relational database engine written in Java and it supports an in-memory table in addition to the disk-based table. The multi-threaded version of this code

critical section (file:line Number)	*coverage phase [%]*	*count*	*size [# Insts]*	*read-set*		*write-set*	
				avg	*total*	*avg*	*total*
Session.java:870	94	50005	12200	283	14155316	49	2489426

Table III. Transaction characteristics in hsqldb.

	throughput [tx/sec]
Lock-HSQLDB (original)	22154
STM-HSQLDB	4417 (5x smaller)

Table IV. HSQLDB throughput on a single thread

does not scale well since a single lock controls access to the database object, thus threads serialize their execution at the level of SQL statements. The lock that protects the database is implemented using Java's `synchronized` mechanism in the form of a single `synchronized` block.

Our transactional-memory implementation of HSQLDB transforms the `synchronized` block into a transactional block. This is possible because we use the in-memory variant of the database and therefore no I/O operations are performed within the transaction. This transformation is similar with the transformation done by Chung et al [7].

One of the core data structures in HSQLDB is a B-tree. Although we expected most of the conflicts to occur on the B-tree object, many conflicts occurred on other data structures, such as global counters. We observed that every transaction incremented several shared counters, which caused frequent conflicts among transactions and transaction retries. It was possible to modify the code such that separate counters are used in each thread, thus avoiding conflicts. As a result, most conflicts among transactions in the modified version of the code occur now due to conflicting data accesses to the B-tree.

Table III shows the transactional characteristics in *hsqldb*. Despite the considerable average size of the transactions, only 283 locations are read and 49 locations are written transactionally. For this benchmark, we also determined the average transaction size including STM read and write barriers; despite the sparse instrumentation, the average transaction size amounted to 72346 instructions, i.e., the code attributed to read and write barriers inside the transaction inflates the instruction count of the vanilla critical section by almost a factor of 6.

5.3.1. Compiler evaluation

Table IV shows that the throughput of HSQLDB with STM on a single thread is five times smaller than that of the original HSQLDB. Figure 6(a) shows the execution time breakdown of HSQLDB on STM. Most of the time is spent in the STM library: about half of the time is consumed by *stm_read* and by the call to *get_write_entry* that searches the address in the internal write buffer. 8% of time is consumed by *stm_status*, which is called before every barrier to check whether it is in a transaction context since an instrumented method can be called outside the transaction.

Figure 6(b) shows the effectiveness of the optimizations we implemented. Most of the benefit for transaction-local object optimization comes from the elimination of write-barriers

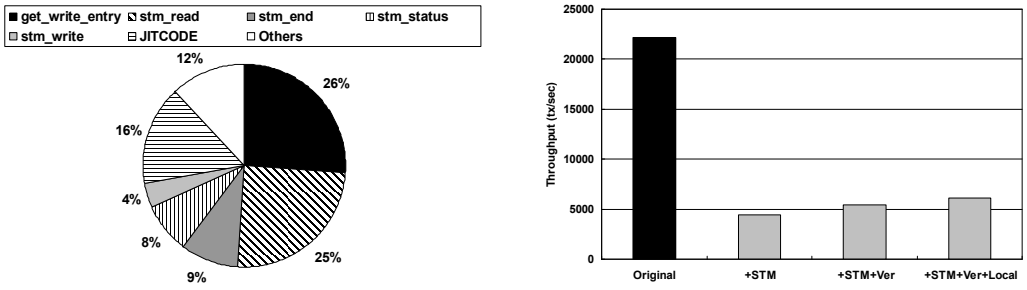


Figure 6. (a) Breakdown of execution time and (b) effectiveness of optimizations (ver: versioning, local: transaction local object optimization)

		STM	*STM+ver+local*
read-set	*avg.*	283	296 (4.5% increased)
	max.	696	748 (8% increased)
write-set	*avg.*	49	39 (20% reduced)
	max.	230	167 (19% reduced)

Table V. Sizes of read and write-set in HSQLDB.

in the constructor. Table V shows that the size of write-sets was reduced by 20% after the optimization. That, in turn, reduces the search time spent in *get_write_entry*, thus speeds up read-barriers. By versioning the instrumented procedures, we significantly reduce calls to *stm_status* (to one per method invocation), and also speedup the execution when a function is not invoked in a transaction context.

Figure 7 shows the performance of hsqldb on an 8-processor Power5 multiprocessor[§]. We observe that the lock-based version of the application does not scale (i.e., with flat throughput) even with more threads. The STM-based version of the implementation scales very well. Although it starts with higher overhead, it outperforms the lock-based version beyond 4 threads.

[§]System administration issues prevented us from setting up the Java STM environment on the 16-processor multiprocessors.

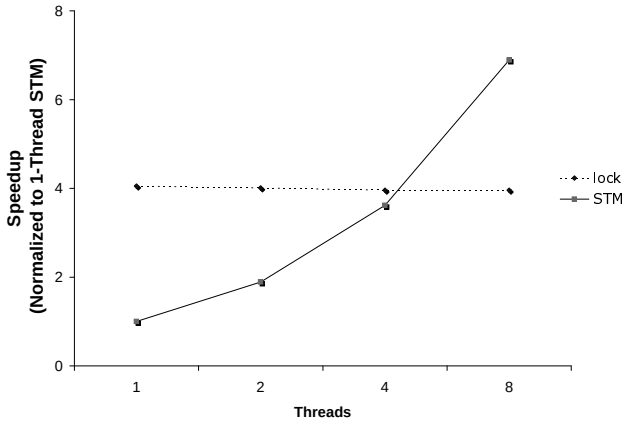


Figure 7. Scalability of hsqldb on an 8-way Power5.

6. Related Work

Due to the vast amount of recent work related to transactional memory, we restrict this discussion to those papers that describe modifications to compilation and managed runtime environments, since that is most directly relevant to the work presented here.

The most closely related work to our C/C++ compiler optimization is the recent release of a C/C++ compiler distribution by Intel that includes support for transactional programs targeting an STM interface, documented by Wang et al [34]. Their system uses annotations to mark transaction boundaries and identify functions that may be called from a transaction, for which the compiler generates a transactional version of the function. Their optimizing compiler includes transactional primitives in its intermediate representation, which is then optimized using partial redundancy elimination, reducing some unnecessary read barriers. The compiler also includes support for code motion across transaction boundaries (when safe), accommodated by an efficient compensating checkpoint mechanism. In contrast, our modifications to XLC are focused on reducing the number of read-/write-barriers using escape analysis, and should be orthogonal to the optimizations implemented in the Intel compiler.

Several Java-based STM systems have also been described. Table VI shows a summary of existing Java-based STM systems. Herlihy et. al. have proposed APIs to define transactions and transactional objects which need to be instrumented [15, 16]. The advantage of this approach is to reduce the number of read-/write-barriers by restricting the instrumented objects with the developer's effort. The disadvantage is that developers need to make much effort to write transactional programs by defining transactional objects in addition to defining transactions.

Harris et. al. and Adl-Tabatabai et. al. have introduced high-level constructs to define transactions [1, 13]. Read-/write-barriers are automatically inserted by the JIT compiler.

Table VI. Summary of Java-based STM systems

	How to define transactions	How to instrument the code
DSTM [16, 15]	Use API: A critical region is defined by implementing a method of a new API.	Use API: An instrumented object is defined by implementing the methods of a new API.
WSTM [13], McRT-STM [1]	Use construct: A critical region is defined by using a new construct.	Use JVM and JIT compiler: Read-/write-barriers are automatically inserted.
Transactional monitor [35]	Use synchronized construct: A synchronized block is recognized as a critical region.	
Azul's optimistic locking mechanismy [31]	Use synchronized construct: A synchronized block is recognized as a critical region.	Use JVM and hadware support: A lock is acquired by multiple threads and the data contention is detected by the hardware.

Although this approach reduces the work of developers, it increases the number of read-/write-barriers. Adl-Tabatabai et. al. describe some optimization techniques to reduce the overhead of each read-barrier by employing the eager-write policy (in which it is not necessary to search to-be-written values), and to reduce the number of read-/write-barriers by employing the object-based conflict detection with some compiler optimizations. The compiler optimizations they proposed are: 1) versioning, 2) barrier elimination of transaction local objects, 3) redundant barrier elimination using the traditional redundancy elimination technique, and 4) barrier inlining. Two optimizations mentioned in 4.2 are a part of those optimizations.

There has also been recent work leveraging transactional hardware primitives in support of optimizations that require efficient conflict detection and/or checkpointing[23, 33]. In contrast to the work presented here, which focuses on reducing the overheads of transactional runtime systems, these papers explore compiler and runtime optimizations that leverage TM hardware to optimize applications regardless of whether they may have been written to use transactions.

7. Conclusions and Future Work

This paper presents our preliminary experience adding support for transactions to an industrial strength compiler and managed runtime. Using several multithreaded applications, we have studied the overheads of executing these applications on a software transactional memory system runtime. Based on the characterization of the execution we designed and implemented several optimizations attacking these overheads. By exploiting the synergy between the design and implementation of the STM and the ability of the compiler to analyze and optimize transactions we obtained speedups of up to 46% (32% average over 1–16 threads runs on the B+tree application).

While respectable, significantly more work needs to be done to eliminate the conflict-free overheads of STMs. We believe that this is certainly within the realm of the infrastructure that we built. It should also be noted that although this work has largely focused on reducing the

single-thread overheads caused by TM, given an application with enough inherent parallelism and enough processors on which to run, these overheads are constant per-thread, and will ultimately be overcome by the additional concurrency exposed by transactional memory. Examples of such potential are shown in our results on the HSQLDB application, which scales almost linearly using transactions while it provides no scaling on the lock based implementation.

Furthermore, beside the performance advantages of transactional memory, transactional programming has the potential of improving programmer productivity. Indeed, one such example is demonstrated in this paper, since we used the compiler to instrument all the read and write accesses in our applications, and the programmer had to just mark the transactional blocks. However, transactional memory must be better integrated in a parallel programming model to clearly demonstrate its potential.

REFERENCES

1. 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 *Proc. of PLDI '06*, pages 26–37, 2006.
2. C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 316–327. Feb 2005.
3. D.A. Bader and K. Madduri. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. In *Proc. 12th International Conference on High Performance Computing (HiPC 2005)*, pages 465–476, December 2005.
4. Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 24–34, New York, NY, USA, 2007. ACM Press.
5. Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Proc. of ISCA 2006*, pages 237–238, 2006.
6. Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 347–358, New York, NY, USA, 2006. ACM Press.
7. JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olkotun. The common case transactional behavior of multithreaded programs. In *Proc. of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, pages 266–277, 2006.
8. Intel Corp. Intel C++ STM compiler, Prototype Edition, September 2007.
9. Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM Press.
10. David Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *DISC*, pages 194–208, Sep 2006.
11. Robert Ennals. Efficient Software Transactional Memory. Technical Report IRC-TR-05-051, Intel Research Cambridge Tech Report, Jan 2005.
12. Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004.
13. Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. In *Proc. of the OOPSLA'03*, pages 14–25, 2006.

14. Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing Memory Transactions. In *Proc. of the PLDI'06*, pages 388–402, 2003.
15. Maurice Herlihy, Victor Luchangco, and Mark Moir. A Flexible Framework for Implementing Software Transactional Memory. In *Proc. of the OOPSLA'06*, pages 253–262, 2006.
16. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proc. of the ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.
17. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architecture support for lock-free data structures. In *Proc. of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.
18. hsqldb - 100 % java database. http://hsqldb.org/.
19. P. L. Lehman and S. Bing Yao. Efficient Locking for Concurrent Operations on B-trees. *ACM Transactions on Database Systems*, (4):650–670, 1981.
20. Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive Software Transactional Memory. In *Proc. of the 19th International Conference on Distrubted Computing*, pages 354–358, September 2005.
21. Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the Overhead of Software Transactional Memory. In *ACM SIGPLAN Workshop on Transactional Computing*. Jun 2006.
22. Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. Feb 2006.
23. Naveen Neelakantam, Ravi Rajwar, Suresh Srinivas, Uma Srinivasan, and Craig Zilles. Hardware Atomicity for Reliable Software Speculation. In *Proc. of ISCA '07*, pages 174–185, 2007.
24. Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505. IEEE Computer Society, Jun 2005.
25. Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *DISC*, pages 284–298, 2006.
26. 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 *Proc. of PPoPP'06*, pages 187–197, March 2006.
27. Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural Support for Software Transactional Memory. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, pages 185–196, December 2006.
28. Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proc. of the Symposium of Principles of Distributed Computing*, pages 204–213, 1995.
29. Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steve Balensiefer, Dan Grossman, Richard Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in stm. In *Proc. of PLDI '07*, pages 78–88. 2007.
30. Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *DISC*, pages 179–193, 2006.
31. Azul Systems. Optimistic Thread Concurrency: Breaking the Scale Barrier. White Paper, January 2006.
32. Mark Tremblay. Transactional Memory for a Modern Microprocessor. 2007 Conference on Scalable Approaches to High Performance and High Productivity Computing, September 2007.
33. Christoph von Praun, Luis Ceze, and Calin Cascaval. Implicit parallelism with ordered transactions. In *Proc of PPoPP 2007*, pages 79–89, 2007.
34. Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *Proc. of CGO '07*, pages 34–48, 2007.
35. Adam Welc, Antony L. Hosking, and Suresh Jagannathan. Transparently Reconciling Transactions with Locking for Java Synchronization. In *ECOOP 2006: European Conference on Object-Oriented Programming, volume 4067 of Lecture Notes in Computer Science. Springer-Verlag.*, pages 148–173, 2006.
36. Luke Yen, Jayaram Bobba, Michael M. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture(HPCA)*. Feb 2007.

