LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-422888

A Case for Including Transactions in OpenMP

M. Wong, B. L. Bihari, B. R. de Supinski, P. Wu, M. Michael, Y. Liu, W. Chen

January 26, 2010

International Workshop on OpenMP
Tsukuba, Japan
June 14, 2010 through June 16, 2010

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

A Case for Including Transactions in OpenMP

Michael Wong[1], Barna L. Bihari[2], Bronis R. de Supinski[2],
Peng Wu[1], Maged Michael[1], Yan Liu[1], Wang Chen[1]

[1] IBM Corporation [2] Lawrence Livermore National Laboratory
{michaelw,yanliu,wdchen}@ca.ibm.com
{bihari1,bronis}@llnl.gov
{pengwu,magedm}@us.ibm.com

Abstract. Transactional Memory (TM) has received significant attention recently as a mechanism to reduce the complexity of shared memory programming. We explore the potential of TM to improve OpenMP applications. We combine a software TM (STM) system to support transactions with an OpenMP implementation to start thread teams and provide task and loop-level parallelization. We apply this system to two application scenarios that reflect realistic TM use cases. Our results with this system demonstrate that even with the relatively high overheads of STM, transactions can outperform OpenMP critical sections by 10%. Overall, our study demonstrates that extending OpenMP to include transactions would ease programming effort while allowing improved performance.

1 Introduction

Many have observed that Transactional Memory (TM) could simplify shared memory programming substantially by simply marking a group of load and store instructions to execute atomically, rather than using locks or other synchronization techniques. TM's promise of easier program understanding, along with composability and liveness guarantees has led to a fad status for TM. As a result, extensions to OpenMP [6] to include transactions have received interest in the OpenMP community [5,8]. However, we must first determine whether TM can be more than just a research toy before these extensions can receive serious consideration.

Two implementation strategies are available for TM. Hardware TM (HTM) modifies the memory system, typically through modifications to the L1 cache, to support atomic execution of groups of memory instructions. Software Transactional Memory (TM) provides similar functionality without using special hardware. In this paper, we combine an STM system that provides the transaction primitive with an OpenMP implementation that provides all other shared memory functionality. The combination is natural since the TM system relies on compiler directives that are similar to OpenMP's syntax.

The remainder of this paper is organized as follows. We present our STM system in Section 2. We then review its integration with a production-quality OpenMP compiler in Section 3. In Section 4, we present performance results for two application scenarios, which demonstrate that transactions can improve performance by 10% over a production quality critical section implementation even with the relatively high overhead of STM. Overall, we conclude that extending OpenMP to include transactions would reduce programming effort while supporting potential performance gains.

2 The IBM XL STM Compiler

Generally, an STM system uses a runtime system to manage all transactional states. This runtime annotates reads and writes for version control and conflict detection. If two transactions conflict, (i. e., the write set of one intersects with the read set of the other), the system may delay or abort and retry one of them. The system validates the reads at the end of the transaction (i. e., any loaded values have not changed). The system then commits the writes (i. e., stores them to their actual memory locations) if it does not detect any conflicts. Compared to HTM, STM does not require special hardware while enabling scalability of inherently concurrent workloads at the cost of higher overhead, particularly when no conflicts occur.

IBM released a freely downloadable STM compiler under alphaWorks® site [7] based on its production IBM® XL C/C++ Enterprise Edition for AIX®, Version 9.0, called IBM XL C/C++ for Transactional Memory for AIX in 2008. This implementation includes standard and debugging versions of the runtime libraries. Fig. 1 shows the TM features [4] of our XL STM compiler.

STM characteristics	IBM STM compiler
Isolation	Weak
Granularity	8 byte block
Direct/Deferred Update	Deferred or Lazy
Concurrency Control	Optimistic
Synchronization	Blocking
Conflict Detection	Early(read after write) Late (write after write)
Inconsistent Reads	Tolerated(signals and infinite loop checks)
Conflict Resolution	Polite
Nested Transaction	Flat
Exception	terminate

Fig. 1. Transactional features of XL STM

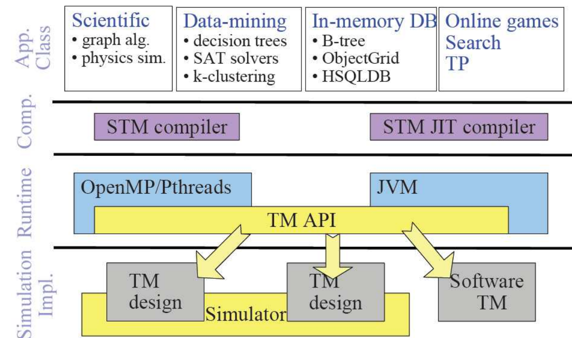

Fig. 2. Project Components of our STM that rely on OpenMP

Our STM uses a block-based mapping of shared memory locations, which enables support for different languages, unlike the alternative object-based mapping. The system buffers writes, which are written to the global address space only when the transaction is guaranteed to commit. When a transaction writes to a stack location or 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. These contention-free writes do not require write-barriers, but may need to be memory checkpointed if the system must recover the overwritten values upon a retry [9].

Using data-flow analysis, the compiler can exclude writes to most contention-free locations from memory checkpointing. Basically, the write requires no checkpointing if the 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. However, contention-free writes often have uses after exiting the transaction. In this case, we can still avoid checkpointing if the location is not live upon entry to the transactional scope the write to the location dominates the transaction end. The latter property guarantees that the location will always be re-defined upon retry and, thus, we do not 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.

The read barrier does not write to shared meta-data. The checkpoint barrier records the original value for writes to contention-free locations. The compiler implements retries with `setjmp` and `longjmp`. In addition, our STM system focuses on instrumentation statistics to identify STM bottlenecks in applications, thus avoiding time consuming and error-prone manual instrumentation.

We use C to implement the runtime system, the source code of which is freely available as part of an Open Source Amino project [2]. This runtime supports exploration of a range of TM scenarios (HTM, STM and hardware accelerated STM) in multiple languages including C, C++ and Java^{TM} as Fig. 2 shows. The current implementation assumes *weak isolation*: accesses to a particular shared location always occur within transactions (a *transactional location*) or never within them. Further, it assumes that transactions only include any revocable operations without side effects (e. g., no file I/O) and, hence, we can safely undo and retry them.

Our runtime uses metadata to synchronize transactional access to shared memory locations. We associate a metadata entry with transactional location. This entry includes a version number for tracking updates of the location and a lock to protect updates of it. A thread can write to memory only if it holds the associated metadata lock. A transaction increments the version number when it releases the metadata lock, which guarantees that the data has not changed if the version number is unchanged. Thus, a transaction can read a metadata version number and then the associated data and then later check the version number to determine if the data is unchanged.

Our runtime maintains information for each thread that includes its read set, write set, statistics, status, level of nesting, and lists of mallocs, frees and modified local variables. The read set information is the metadata location and version number. The write set information contains the address, value, size and metadata location.

When a thread begins a transaction, it sets its transactional status data. It also reads some global data in some configurations. The thread then records the current version number before it reads a transactional location. In some configurations, the thread also checks the consistency of the read set. For transactional writes, the thread records the target address and the value to be written. In some configurations, it acquires the corresponding metadata lock; otherwise, it acquires metadata locks for all locations in its write set at the end of the transaction. When the transaction ends, the thread then validates the consistency of its read set. If that fails, the thread aborts the transaction by releasing the metadata locks and jumping to the beginning of the transaction. Otherwise, the thread writes the values to the addresses in its write set, releases the metadata locks and then resets its transactional status data.

The STM runtime can collect statistics related to a program's inherent transactional characteristics (i. e., independent of our STM implementation), such as trans-

Statistic	**Description**
READ_ONLY_COMMITS	Number of committed transactions with no writes
READ_WRITE_COMMITS	Number of committed transactions with writes
TOTAL_COMMITS	Number of successfully committed transactions
TOTAL_RETRIES	Number of retried transactions
AVG_RETRIES_PER_TXN	Average number of retries per committed transaction
MAX_NESTING	Maximum level of transaction nesting
READ_SET_SIZES	Unique locations in read sets of committed transactions
WRITE_SET_SIZES	Unique locations in write sets of committed transactions
AVG_READ_SET_SIZE	Average number of unique locations in read set per transaction
AVG_WRITE_SET_SIZE:	Average number of unique locations in write set per transaction
READ_SET_MAX_SIZE	Maximum number of unique locations in a read set
WRITE_SET_MAX_SIZE	Maximum number of unique locations in a write set
READ_LIST_MAX_SIZE	Maximum number of locations in a read list
WRITE_LIST_MAX_SIZE	Maximum number of locations in a write list
DUPLICATE_READS	Number of transactional reads of locations previously read in the same transaction
PCT_DUPLICATE_READS	Percentage of transactional reads of locations previously read in the same transaction
DUPLICATE_WRITES	Number of transactional writes to locations previously written in the same transaction
PCT_DUPLICATE_WRITES	Percentage of transactional writes to locations previously written in the same transaction
NUM_SILENT_WRITES	Number of transactional writes of already stored value
PCT_SILENT_WRITES	Percentage of transactional writes of already stored value
READ_AFTER_WRITE_MATCHES	Number of transactional reads that follow a transactional write of the same location in the same transaction
PCT_READ_AFTER_WRITE	Percentage of transactional reads that follow a transactional write to the same location in the same transaction
NUM_MALLOCS	Number of calls to `malloc` inside transactions
NUM_FREES	Number of calls to `free` inside transactions
NUM_FREE_PRIVATE	Number of calls to `free` blocks allocated in that transaction

Table 1. STM runtime statistics

action sizes. It can also track implementation specific data, such as metadata locks acquired. We collate these statistics by static transactions (i. e., source code file name and first line number). We also generate aggregate statistics. Collecting these statistics incurs a significant performance cost but can guide optimization of TM programs.

The program must call `stm_stats_out` in order to generate the statistics files. We allow multiple calls to `stm_stats_out`. The statistics can be inconsistent if the call occurs while any transactions are active since the statistics can change during the snapshot. We recommend only calling `stm_stats_out` when only the main thread is active, which ensures the statistics are consistent. The `stats.h` file has a full list of the statistics; we provide some of the most important ones in Table 1.

We have ported our STM runtime to several platforms including AIX and Linux® on IBM PowerPC®, LinuxX86, and LinuxX86_64, in 32 or 64bit mode. IBM XL

C/C++ for Transactional Memory for AIX generates code for this interface for programs that use the high-level interface that we discuss in the next section. Other compilers could also use our open source STM runtime for this interface.

3 IBM STM Compiler design

3.1 Syntax

Programs define transactions for our STM compiler [3] through a simple directive:

```
1 #pragma tm atomic [default(trans|notrans)]
2   {
3
4   }
```

in which `default(trans|notrans)` defines the default behavior of memory referenced in the lexical scope of the transactional region. If the user specifies `default(trans)` then the compiler translates references to shared variables in the transactional region to STM runtime calls, often referred to as STM read-write-barriers, that ensure the correctness of the execution. If the user specifies `default(notrans)` then the compiler does not translate any memory references in the region to STM read-write-barriers. The `default` clause is optional; the default value is `trans`.

The code region encapsulated within our transactional construct is basically a structured block although the block cannot include any OpenMP constructs. We impose this restriction since otherwise an aborted transaction could lead to an inconsistent state for the OpenMP runtime library, such as acquiring a lock that it will never release.

Fig. 3 shows an example transactional region within an OpenMP parallel region. The compiler inserts `stm_begin()` and `stm_end()` around the transactional region. These calls allow our STM runtime to monitor all shared memory access within the transactional region in order to ensure the transaction executes correctly.

3.2 Special Transactional Function Attributes

Users must annotate functions that are called within transactional regions so that the compiler can correctly transform memory references within the function to STM runtime barrier calls. We provide two function attributes for this purpose: `tm_func` and `tm_func_notrans`. The user specifies `__attribute__((tm_func))` with a function declaration to indicate that a transactional region can call the function so the compiler must transform its memory references to STM runtime calls. The user specifies `__attribute__((tm_func_notrans))` with a function declaration to indicate that transactional region can call the function but the compiler should not transform its memory accesses to STM runtime calls. Neither attribute is required if transactional regions cannot call the function. Fig. 4 shows an annotated function declaration.

Our transactional function attributes reflect two key design factors. First, they allow function calls within transactions without requiring the user to make major code modifications. Second, the attributes allow the compiler to check that the function call conforms to our restriction that transactional regions cannot include any OpenMP constructs. The compiler issues a warning for any unannotated functions that are called

```
1   int b[25], j;
2   int index[5] = {4,5,675,22,34};
3
4   for( j = 0 ;j<25; j++ )
5      b[j] = 0;
6
7   #pragma omp parallel for
8      {
9      for(j=0; j<25; j++)
10        {
11        ...
12
13        #pragma tm atomic
14           {
15           b[index[j]] = ...;
16           }
17        }
18     }
19  }
```

Fig. 3. Sample code of using #pragma tm atomic

```
1 int foo (int sum) __attribute__((tm_func));
2 int foo (int sum)
3 {
4 return ++sum;
5 }
```

Fig. 4. Sample code of annotating function attribute to a function.

within a transactional region. The programmer must ensure that the call is safe with the default behavior that does not transform memory references to STM runtime calls. These attributes are only needed for STM; HTM implementations do not require them.

4 Experimental Results

We provide results that evaluate the potential for TM to benefit scientific computing. In particular, we consider the opportunities to use transactions in unstructured-mesh multi-physics simulation applications, which are widely used because of their geometric and architectural flexibility. These applications typically have many compute-intensive loops with complicated memory referencing patterns. Although these applications usually exhibit good scalability with MPI-based domain-decomposition, the trend toward systems built with multicore nodes motivates explorations of moving them to a hybrid OpenMP/MPI programming model. However, many users view the complexity of shared memory programming in general, and of ensuring data race freedom in particular, as a significant barrier. Transactions directly address this concern.

While no production HTM implementations currently exist, STM provides a mechanism to experiment with using transactions within these applications. Thus, we have implemented a Benchmark for Unstructured-mesh Software Transactional Memory (BUSTM). BUSTM provides a simple code in which to explore the programming benefits of trans-

```
1 #pragma tm atomic default(trans)
2 {
3   gradient[cell_no_1] += incr;
4   gradient[cell_no_2] -= incr;
5 }
```

Fig. 5. Gradients accumulated within transaction in `compute_cell`

actions and any performance implications as it mimics the algorithms and behavior of real unstructured mesh applications.

When we construct a benchmark of an application scenario, we must ensure that it captures the salient features of the target application space. Since we primarily focus on race conditions and the potential benefits of TM, we artificially generate memory conflicts in either a deterministic or random yet still controllable manner. That is, even with random conflicts, we can configure their probabilities indirectly through input parameters. The benchmark also must include rigorous error checking for example problems with known or independently computable answers to ensure that the threaded experiments (with or without transactions) execute correctly.

BUSTM meets these requirements and uses realistic unstructured mesh connectivity. It mimics the complex and flexible bookkeeping common to unstructured mesh applications. BUSTM can handle unstructured cells with an arbitrary number of faces. We have already used a wide range of cell types, including triangular prisms, hexahedra, tetrahedra and pyramids. Like real unstructures mesh applications, BUSTM cross references the basic unstructured mesh building blocks of **nodes**, **faces** and **cells** in almost all combinations, so that *indirect indexing* pervades the code. This indirect addressing leads to extensive synchronization requirements to use shared memory programming, which is exactly when TM should provide benefits. The remainder of this section explores the benefits of our STM implementation with both deterministic and probabilistic conflicts.

4.1 Deterministic Conflicts

Domain decomposition based on MPI message passing has provided successful parallelization of conservative finite volume schemes on unstructured grids. Our careful evaluation of their typical memory access patterns, however, found that conflicts may occur as cells are updated during the traversal of face-based loops. Although these conflicts rarely occur in practice, we cannot assume they do not occur. Since ignoring them would lead to incorrect results, we must protect them with some synchronization method. However, locks have relatively high overhead when conflicts are unlikely. Thus, TM may provide substantial performance benefits for this scenario.

Short of a full-blown CFD solver, we simulate face-by-face flux computations with the *numerical divergence* of a mesh-function that we define on an unstructured mesh in a cell-centered sense. This emulation computes the gradient of a function. If the function is constant, its gradient and, thus, divergence is zero. Fig. 5 shows the transaction.

Since real finite-volume codes have significant computation per face, BUSTM loops over the faces, as Fig. 6 shows. Memory conflicts can occur if different faces update the

```
1 #pragma omp parallel for
2     for (i=0; i < max_face; i++){
3       left_neighbor = left_cells[i];
4       right_neighbor = right_cells[i];
5       compute_cell(incr, left_neighbor);   //face increments left cell
6       compute_cell(incr, right_neighbor); //face increments right cell
7     }
```

Fig. 6. Threaded face loop that calls `compute_cell`

same cell. They only actually occur, resulting in incorrect execution, if `compute_cell` does not use transactions or other synchronization such as a `critical` section, and two updates happen at the same physical time. Thus, the probability of conflicts is extremely low and many of our experimental runs had no conflicts.

Our experiements use a mesh of 119893 triangular prism cells arranged as a 2-D layer of 3-D cells. This mesh has 420060 faces (some quadrilateral, others triangular) and 123132 nodes. The total number of potential conflicts is fixed with this mesh although the actual number of conflicts varies between runs. We observe the number of resolved conflicts from the statistics available with our STM implementations.

Fig. 7 shows the number of conflicts resolved in each of 1000 runs is always less than ten and frequently zero. Fig. 8 shows the sum of the number of conflicts that our STM implementation resolves over 1000 runs versus the total number of errors committed (without STM) over 1000 runs for a range of thread counts. We observe many fewer STM retries than errors with the unsynchronized code. However, both exhibit similar trends with increasing thread counts and rarely occur relative to the number of potential conflicts. Despite the relatively small mesh (which *increases* the conflict probability), only 0.00042% of all cell updates incurred conflicts on 16 threads. No conflicts occur with just two threads (or one, which is clearly expected), which makes debugging the unsynchronized code more difficult. The number of conflicts increases significantly with the number of threads, with conflicts being fairly likely with 16 threads.

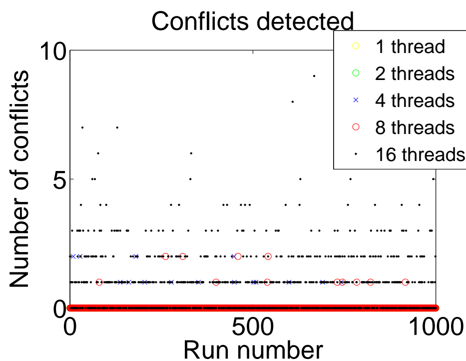

Fig. 7. Resovled deterministic conflicts

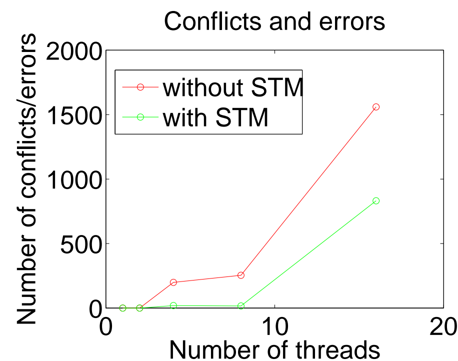

Fig. 8. Total conflicts or errors

```
1 #pragma tm atomic default(trans)
2 {
3   cell_counter[cell_no] ++;
4 }
```

Fig. 9. Cell counter incremented within transaction of `mark_cell`.

4.2 Probabilistic Conflicts

Monte Carlo applications are another common type of scientific application in which memory conflicts potentially occur with threaded implementations. In these applications, randomly released particles travel through a computational mesh and increment a cell-based physical quantity each time they touch a cell. Parallelizing over the particles results in almost embarassingly parallel loops, except for the race condition produced by two particles (belonging to two different threads) trying to update the same cell at the same time. Although these memory conflicts are unlikely, some will occur with enough particles and threads.

We exploit the unstructured bookkeeping in BUSTM in order to emulate the behavior of particles without implementing a real Monte Carlo application. Instead of particles that travel along a straight line through space, they travel from cell to cell via the neighbor information available for each adjacent cell. Thus, our benchmark has two levels of randomness. First, we randomly select the cell in which the particle is "born". Second, we randomly choose the face that the particle exits the current cell.

We make sure the number of particles is independent of the number of cells. After being created, or "born," each particle is "alive" as long as it stays within the computational domain (i. e., the face through which it exits the current cell is an interior face). If that randomly selected face is a boundary face, it exits the domain and completes its path. This scheme results in a wide variance in the particle path lengths; some will have a short lifespan while others stay active within the domain for a long time. This property, which is consistent with real Monte Carlo simulations, limits scalability.

Fig. 9 shows the simple transaction that safely increments a single cell-based integer. Fig. 10 shows the loop that distributes the particles across the threads. As discussed above, each randomly generated particle moves through the mesh until it exits the domain. We also increment a separate counter for each particle so that we can compute the total number of touches without concern for potential conflicts as `cell_counter*particle_counter`. This check usually fails if we do not use any synchronization for the `mark_cell` calls.

Our probabilistic experiments use the same triangular prism mesh as the deterministics ones and we report conflict statistics observed by our STM implementation. We performed 100 runs, using 12000 random particles (10% of the number of cells) during each run. Fig. 11 shows a much higher number of conflicts than in the deterministic case. The conflicts are fairly consistent for a given thread count and appear almost linearly proportional to that count. We observe the opposite trend from the deterministic case between the total number of STM resolved conflicts and the total number of unsynchronized errors for this probabilistic test, as Fig. 12 shows. We find that STM incurs

```
1 #pragma omp parallel for
2 for(i=0; i<max_particles; i++){
3   next_cell = rand();
4   while(inside){
5     mark_cell(next_cell);   //particle increments cells
6     next_face = rand();
7     next_cell = neighbor(next_face);
8     if(next_cell < 0)inside = 0;
9   }
10 }
```

Fig. 10. Threaded particle loop that calls `mark_cell`

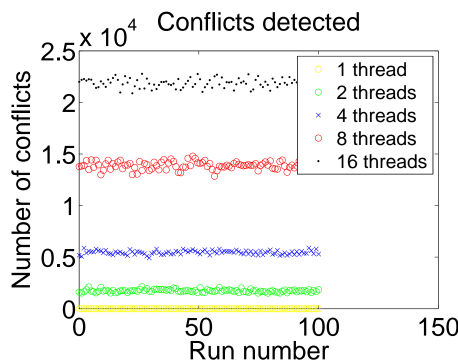

Fig. 11. Resolved probabilistic conflicts

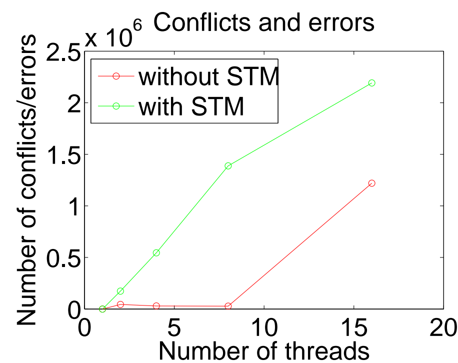

Fig. 12. Total conflicts or errors

many more conflicts, sometimes by an order of magnitude. While we are still investigating this discrepancy, the much heavier computational load imposed by the frequent invocation of a random number generator may cause it. Nonetheless, the total number of resolved conflicts and committed errors is still small compared to the number of updates. For example, the conflict probability is only 0.009% on this relatively small mesh with 16 threads.

While the deterministic and probabilistic algorithms represent very different numerical algorithms and in their relationships of resolved conflicts to errors, they also have similarities. In both cases we have low conflict probabilities (much less than 0.01%) and both use the same unstructured mesh. Indeed, the results confirm our conjecture that the algorithms are well suited to transactions since conflicts rarely occur. We can observe reasonably good performance even with STM and expect higher performance with HTM since conflict resolution is generally the dominant cost with TM. Lock based implementations, on the other hand, would suffer much more overhead. Preliminary timing results that compare our STM implementations to ones that use the OpenMP `critical` construct indicate that STM provides about a 10% performance advantage despite the relatively large overheads of STM [1].

5 Conclusions and Future work

We have presented the design and implementation of a software transactional memory system. Our system combines an open source runtime with modifications to the production IBM® XL C/C++ Enterprise Edition for AIX®, Version 9.0 compiler. This system uses OpenMP to generate threads and parallelize applications. Transactions denoted by a simple directive serve as an alternative to OpenMP synchronization. Users also must annotate declarations of any functions accessed within a transaction. These annotations would be unnecessary with HTM support since they direct the compiler to transform memory accesses to STM primitives.

We evaluate the efficacy of TM for scientific computing. In particular, we develop a new TM benchmark, BUSTM, that explores the use of transactions for unstructured mesh applications. We present two distinct scenarios that BUSTM can emulate: CFD applications and Monte Carlo applications. In both cases, we find that conflicts for the transactions are infrequent; however, correct execution requires synchronization of some sort. Our initial performance results found that the STM implementation outperformed the equivalent OpenMP implementation with `critical` regions by 10%, a significant result considering that STM has relatively high overhead. Although we are continuing to explore these performance results with different meshes, we expect that the emulated scientific applications will benefit substantially from HTM support.

Acknowledgements

The authors wish to thank John Gyllenhaal and Scott Futral of LLNL for numerous fruitful discussions on this subject and for support of this work. The second author also acknowledges past financial support from Rockwell International, Boeing, and Hypercomp, Inc. in developing the unstructured mesh bookkeeping used in the experiments, and from Icon Consulting and IBM in writing the BUSTM code.

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DEAC52-07NA27344 (LLNL-CONF-422888).

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other brands and names are the property of their respective owners.

References

1. B. L. Bihari. Experiments Using IBM's Software Transactional Memory Compiler. http://spscicomp.org/ScicomP15/slides/user/bihari.pdf, May 2009.

2. IBM. Concurrent Building Block. http://sourceforge.net/projects/amino-cbbs/, May 2008.
3. IBM. IBM XL C/C++ for Transactional Memory for AIX, V0.9 Language Extensions and Users Guide. http://dl.alphaworks.ibm.com/technologies/xlcstm/xlcstm-whitepaper.pdf, May 2008.
4. J.R. Larus and R. Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, January 2007.
5. M. Milovanović, R. Ferrer, O. Unsal, A. Cristal, X. Martorell, E. Ayguadé, J. Labarta, and M. Valero. Transactional Memory and OpenMP. In *Proc. of the 3rd Intl. Workshop on OpenMP: Practical Programming Model for the Multi-Core Era*, pages 37–53, Beijing, China, June 2007. LNCS 4935.
6. OpenMP ARB. OpenMP Application Program Interface, v. 3.0, May 2008.
7. M. Wong. IBM XL C/C++ for Transactional Memory for AIX. 2008. http://www-949.ibm.com/software/rational/cafe/blogs/ccpp-parallel-multicore/2009/08/11/ibms-alphaworks-software-transactional-memory-compiler, Aug 2009.
8. B. Woongki, C.C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM Transactional Application Programming Interface. In *PACT'07: Proceedings of the 16th International Conference on Parallel Architec-ture and Compilation Techniques*, pages 376–587, Washington, DC, USA, June 2007. IEEE Computer Society.
9. P. Wu, M. Michael, C. von Praun, T. Nakaike, R. Bordawekar, H. Cain, C. Cascaval, S. Chatterjee, S. Chiras, R. Hou, M. Mergen, X. Shen, M. Spear, H. Y. Wang, and K. Wang. Compiler and Runtime Techniques for Software Transactional Memory Optimization. In *Journal of Concurrency and Computation: Practice and Experience*, pages 7–23, Chichester, UK, July 2009. IEEE Computer Society.