LLNL-CONF-523539

What Scientific Applications can Benefit from Hardware Transactional Memory? - Early experience from a commercially available HTM system.

M. Schindewolf, M. Schulz, B. Bihari, J. Gyllenhaal, A. Wang, W. Karl

January 20, 2012

International Conference on Supercomputing (ICS)
Venica, Italy
June 25, 2012 through June 29, 2012

What Scientific Applications can Benefit from Hardware Transactional Memory?

Early experience from a commercially available HTM system.

ABSTRACT

Achieving efficient and correct synchronization of multiple threads is a difficult and error-prone task at small scale and, as we march towards extreme scale computing, will be even more challenging when the resulting application is supposed to utilize millions of cores efficiently. Transactional Memory (TM) is a promising technique to ease the burden on the programmer, but only recently has become available on commercial hardware in the new Blue Gene/Q system and hence the real benefit for realistic applications has not been studied, yet.

This paper presents the first performance results of TM embedded into OpenMP on a prototype system of BG/Q and characterizes code properties that will likely lead to benefits when augmented with TM primitives. We first, study the influence of thread count, environment variables and memory layout on TM performance and identify code properties that will yield performance gains with TM. Second, we evaluate the combination of OpenMP with multiple synchronization primitives on top of MPI to determine suitable task to thread ratios per node. Finally, we condense our findings into a set of best practices and apply them to a Monte Carlo Benchmark, closely representing a real world application, to optimize its performance. This optimized TM version, executed with 64 threads on one node, yields a speedup of 26.47 over baseline.

General Terms

Performance Analysis, Hardware Transactional Memory

Keywords

Performance Analysis, Hardware Transactional Memory, Early Experience, BG/Q

1. INTRODUCTION

Achieving efficient and correct synchronization of multiple threads is a difficult and error-prone task. In particular, lock-based synchronization schemes often lead to high-overheads, either due to lock contention, when using coarse grain locks, or unnecessary lock overhead, when using fine grain locks. This not only slows down the overall process using the locks, but also has a global effect in large scale programming due to the creation of skew between processes as well as load imbalance, both major factors limiting the scalability of applications.

Transactional Memory (TM) has been proposed almost a decade ago to tackle this issue in shared memory systems [15]. TM simplifies synchronization by providing a simple construct: the programmer wraps the critical instructions in a transaction (also called atomic block). These transactions then are executed optimistically in parallel and conflicting accesses are resolved by a TM run time system. As a consequence only the effects of entire and completed transactions are visible to concurrent threads avoiding the visibility of intermediate memory states.

Except for a few, by now discontinued prototype implementations in research processors, TM has mainly been confined to software solutions and therefore has been burdened with significant runtime overheads severely restricting its applicability in high performance computing. However, the recently introduced Blue Gene/Q (BG/Q) system by IBM for the first time provides Hardware Transactional Memory (HTM) in a commercially available platform. BG/Q is designed as a large scale platform designed for scientific computing workloads. The first full machine will be installed at Lawrence Livermore National Laboratory and will provide more than 1.6 million compute cores with a total of over 6 million hardware threads, making application scalability one of the premier challenges on this machine.

This paper presents the first performance evaluation of the HTM capabilities on BG/Q from the application perspective. Not every lock-based application will be able to benefit from HTM and it is important to understand what code properties lead to efficient executions and, hence, which codes can benefit from a port to HTM. In order to help code developers with this task, we provide a precise evaluation of the strengths and weaknesses of the architecture as well as what is required to map applications to the architecture in an efficient way. In particular, we focus on the the synchronization primitives for parallel programming in shared memory architectures with OpenMP and provide detail benchmark results. Our experiments take into account the application's characteristic (high or low contention), the influence of environment variables, the effects of enlarging transaction sizes, and hybrid parallelization with MPI. We further apply our results to the optimization of a Monte Carlo Benchmark (MCB) that is characteristic for several large scale Monte Carlo simulations and show they can be used to guide deployment of HTM capabilities.

This paper makes the following contributions:

1. We presents the first performance results of HTM combined with OpenMP of the new BG/Q architecture.

2. We study the influence of thread count, environment variables and memory layout on TM performance.

3. We evaluate MPI with OpenMP and multiple synchronization primitives to determine a fitting task to thread ratio for one node.

4. We identifies code properties that are likely to yield performance gains with TM.

5. We condense the findings into best practices that are applied to a realistic Monte Carlo Benchmark code and optimize its performance by tuning the choice of synchronization primitives.

For the latter, an optimized TM version, executed with 64 threads on one node, outperforms the original code and a simple TM implementation and yields a speedup of 26.47 over the baseline.

The remainder of this paper is organized as follows. Section 2 provides background on Transactional Memory in general as well as related work. Section 3 describes our experimental setup, the TM architecture of the BG/Q system, and our benchmark used to determine overheads. Section 4 presents low-level measurements, followed by the lessons learned in Section 5. Section 6 shows how we can use our lessons to add transactions to a Monte Carlo code and to optimize it. Section 7 concludes and presents ideas for future work.

2. BACKGROUND AND RELATED WORK

Transactional Memory has been proposed as architectural support for lock-free data structures in shared memory systems [15]. The core idea is to replace pessimistic synchronization, such as locks, with optimistic synchronization in the form of transactions. Programmers can group updates into transactions and these can be executed concurrently with the rest of program. A runtime system (in hardware or software) detects conflicts between transactions as well as between a transaction and the rest of the program and, if necessary aborts and rolls the effects of the transaction back. As a consequence, the effects of any transaction are seen as if the transaction occurred as one atomic block, providing the necessary synchronization guarantees.

Many different TM designs have been proposed [14]. Software-based approaches [11, 10, 18, 21] (STM) use a Software Transactional Memory library to implement algorithms for the detection and resolution of conflicting memory accesses. Software is very flexible but also comes with inherent overheads [4]. On the contrary Hardware Transactional Memory systems are fast for transactions that fit into the restricted hardware [20, 12, 16]. Further, hybrid approaches, combining hardware and software to accelerate execution and lift the limits of the hardware have been researched [22, 19, 8, 6].

The only paper that described an early experience with a commercial hardware transactional memory implementation published in a major conference, to our knowledge, is by Dice et al. [9]. The paper describes and evaluates the hardware transactional memory feature of SUN's Rock processor [5], which is no longer available. The focus of their paper is on the evaluation of concurrent data structures such as Red Black trees and `Hashtable`, and the construction of a minimum spanning forest [17]. The parallelization of these codes uses threads only. Thus, no experiments are made that estimate the performance of a hybrid parallelization with MPI.

A description of a second commercial HTM implementation can be found in a paper by Dick [7]. The goal is to accelerate the `synchronized` keyword in Java. Thus, no extensions to the language are made and explicit programming with transactions is not possible. Instead a heuristic decides whether to run a critical section as transaction or not.

Most STM papers use STAMP, a benchmark suite for transactional memory research [3]. The codes comprise: Bayesian network learning, gene sequencing, network intrusion detection, K-means clustering, maze routing, graph kernels, a client/server travel reservation system, and delaunay mesh refinement. This covers many application areas in which STM have been used, but do not represent codes from the area of of high performance computing, for which HTM is a promising approach to overcome synchronization overheads and to improve scaling of hybrid thread/MPI codes. In this paper, we therefore focus on a new benchmark explicitly designed to cover this area and present result that demonstrate how HTM can be deployed in HPC.

3. EXPERIMENTAL SETUP

For all following experiments we use an early prototype of BG/Q installed at IBM. TM is available as HTM through IBM's XL C/C++/Fortran Compiler suite for BGQ, which provides new language primitives that allow users to specify transactions.

3.1 Overview of BG/Q's TM Hardware

The BG/Q prototype we had access to contained 32 nodes with 16 cores each. Each core can execute up to four hardware threads. Transactional memory is implemented within the L2 cache, which consists of 16 banks of 2 MB each located across a full crossbar from the 16 multithreaded compute cores. Each L2 cache has a cache line size of 128 Bytes. Memory accesses that can lead to conflicts between transactions, are tracked by the L2 cache, which is a point of coherency. Conflict detection between different transactions is completed in hardware, while conflict resolution is coordinated through the TM software stack. Note that, in addition to TM, the L2 cache also implements an improved set of atomic operations that also target faster and more efficient thread synchronization. Comparisons in the remainder of the paper between TM and atomic operations therefore provide results between two novel and highly optimized schemes. More information on BG/Q's hardware in general can be found in a recent presentation by Haring at Hot-Chips [13].

Name	Description	Contention
None	Threads do not conflict.	No contention.
Adjacent	Adjacent memory addresses are updated.	No to small contention.
Random	Randomly (but repeatable) seeming updates.	High contention.
FirstParts	Only the first parts are updated	Highest contention.

Table 1: Different contention levels in the CLOMP-TM benchmark.

Name	Implementation	Description
Bestcase	—	Bestcase without synchronization.
Serial Ref	—	Serial reference implementation.
Small TM	`#pragma tm_atomic`	Synchronizing each update with a transaction.
Small Atomic	`#pragma omp atomic`	Synchronizing each update with an atomic operation.
Small Critical	`#pragma omp critical`	Synchronizing each update with OpenMP's critical section.
Large TM	`#pragma tm_atomic`	All scatter zone updates in one transaction.
Large Critical	`#pragma omp critical`	All scatter zone updates in one critical section.
Huge TM X	`#pragma tm_atomic`	*X* times *Large TM* in one transaction.

Table 2: Description of synchronization constructs used in CLOMP-TM.

3.2 Application Perspective in BG/Q's TM Software Stack

By default, the TM runtime defaults to a ´lazy´ (or optimistic) conflict detection scheme, at commit time, as the runtime suppresses the hardware from sending conflict interrupts to the conflicted threads. However, applications/users can enable a 'pessimistic detection' scheme by setting the environment variable `TM_ENABLE_INTERRUPT_ON_CONFLICT`. That is, conflict arbitration happens immediately at the time of conflicts. Either scheme needs to be carefully chosen as an already doomed thread, if allowed to run till the end, may cause further spurious conflicts.

The TM runtime also relies on a lazy versioning (i.e., write-back) scheme as all speculative writes are buffered in the multi-versioned cache, until commit time. Strong atomicity (i.e., opacity) is guaranteed unless a thread is running in irrevocable mode. In such case, the thread runs non-speculatively and all writes take affect immediately. The `TM_MAX_NUM_ROLLBACK` environment variable controls when a thread should enter into irrevocable mode. The irrevocable mode is a mechanism that guarantees that a threads makes progress. The contention manager favors an older thread to commit based upon the timebase register value of the thread at the time when speculation starts. Aborting a transaction does not back-off for pre-determined time, rather, a thread retries immediately. The runtime also implements flat nesting whereby commits and rollback are to the outermost TM region. As an additional feature, the runtime monitors the TM behavior of the application and provides the resulting TM statistics to the user. All TM statistics, presented in this paper, are retrieved by this method.

3.3 The CLOMP-TM Benchmark

Since current TM benchmark suites do not provide the necessary coverage for scientific applications, we focus on a new benchmark specifically designed for this purpose, CLOMP-TM[1], which originates from the publicly available CLOMP benchmark [2].

While the original CLOMP aims to quantify overheads due to threading and the specific OpenMP implementation, the CLOMP-TM aims at quantifying and comparing synchronization overheads of multiple synchronization constructs. CLOMP-TM can be configured to resemble the synchronization characteristics of typical scientific applications used in HPC. Thus, performance results of CLOMP-TM enable us to project the performance of these large scale applications.

[1]For the experiments CLOMP-TM version 1.54 is used and will be made publicly available at publication time of the paper.

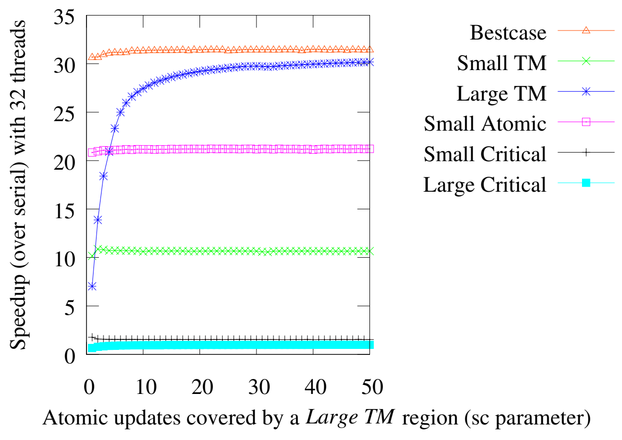
Major changes over CLOMP

Traditionally synchronization is achieved through mutual exclusion. This is deemed a pessimistic synchronization construct because conflicts between threads are prevented through mutual exclusion. Now, Transactional Memory enables optimistic synchronization. Transactions are executed concurrently. The TM run time system monitors transactional memory accesses to detect and resolve conflicts. In case of a conflict one transaction is aborted and its changes are rolled back. This changed behavior has an impact on the application performance and demands to be studied thoroughly. Further, we compare the performance of TM with OpenMP-based constructs with the same level of abstraction in terms of programming.

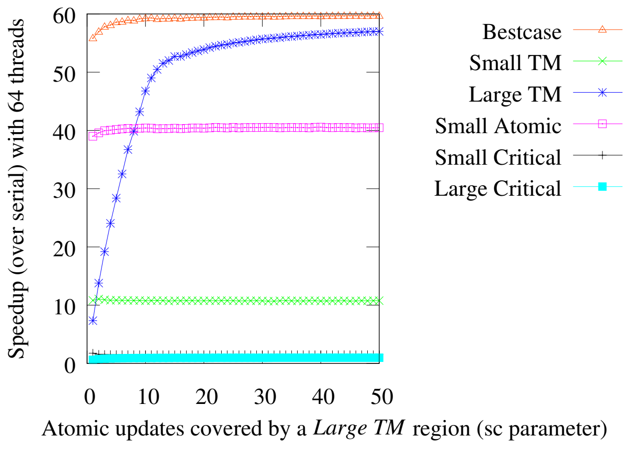
For a meaningful comparison of optimistic and pessimistic synchronization constructs, multiple memory access patterns have to be considered. These memory access patterns determine the likelihood of a conflict between concurrent accesses of two threads. A single parameter defines the zones that are updated by a thread. The contention arises when multiple threads update the same zones. These different contention scenarios are shown in Table 1.

In comparison to the CLOMP benchmark, the updates of a zone are enlarged. This new construct is called "scatter zone" and enables larger critical sections, which resembles the update of multiple variables (e.g., coordinates with multiple dimensions x, y, and z) in one critical section. For the *large* versions of the synchronization constructs, *scatterCount* updates many zone in a single synchronized block.

Each iteration executes the selected computation pattern. Available patterns with increasing complexity are: *none*, *divide*, *manydivide*, and *complex*. CLOMP-TM is carefully designed to eliminate as much noise as possible: I/O is per-



(a) 32 threads.



(b) 64 threads.

Figure 1: CLOMP-TM performing 8 divide operations with a stride of 4 per zone update with excellent speedups of *Large TM* over *Small Atomic*. Run with `clomp-tm-bgq-divide4 -1 1 256 128 256 stride1,1,stride1%/2 sc 1 0 6 100`.

formed only outside of timing loops and all the loops are run just before the timing loops to eliminate start up costs and cold cache effects. Table 2 holds the synchronization constructs to be compared.

4. CLOMP-TM RESULTS

Figure 1 shows an example with CLOMP-TM parameters that we chose to examine the cases where TM outperforms a highly efficient implementation of `omp atomic`. In this configuration CLOMP-TM performs 8 divide operations per zone update with a stride of 4. Threads do not contend for memory locations. We increase the size of the scatter zone so that an increasing amount of updates is carried out in *Large TM*. Figure 1(a) illustrates that in the case of 32 threads performing 4 zone updates is the cross-over point for *Large TM* over *Small Atomic*. For 64 threads the number of zones is twice as high (see Figure 1(b)). Please note that the large amount of computation per zone update masks the overheads of synchronization.

For the CLOMP-TM cases presented in this paper, synchronization overheads can dramatically affect speedup. We vary the parameters of CLOMP-TM to learn how the parameters affect the speedup and to find out what code properties qualify for TM. These results will help application developers to tune their codes (e.g., through picking a better suited synchronization primitive), but the achievable speedup is determined by the properties of the application (e.g., ratio of computation and synchronization, contention for memory locations).

4.1 Synchronization Overhead

We obtain the results in this section by using the parameters shown in Table 3. Memory is allocated by the main thread. This is sufficient because memory access are uniform in BG/Q. The setting of *zonesPerPart* equal to 100 stems from the original CLOMP and mimics the loop sizes of many multiphysics applications [2]. The chosen computation pattern is *divide*. For each zone update 8 extra *divide* calculations are executed. The environment variable

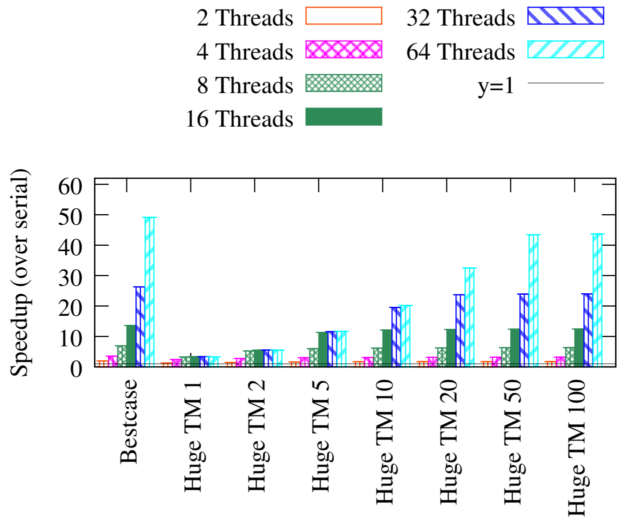
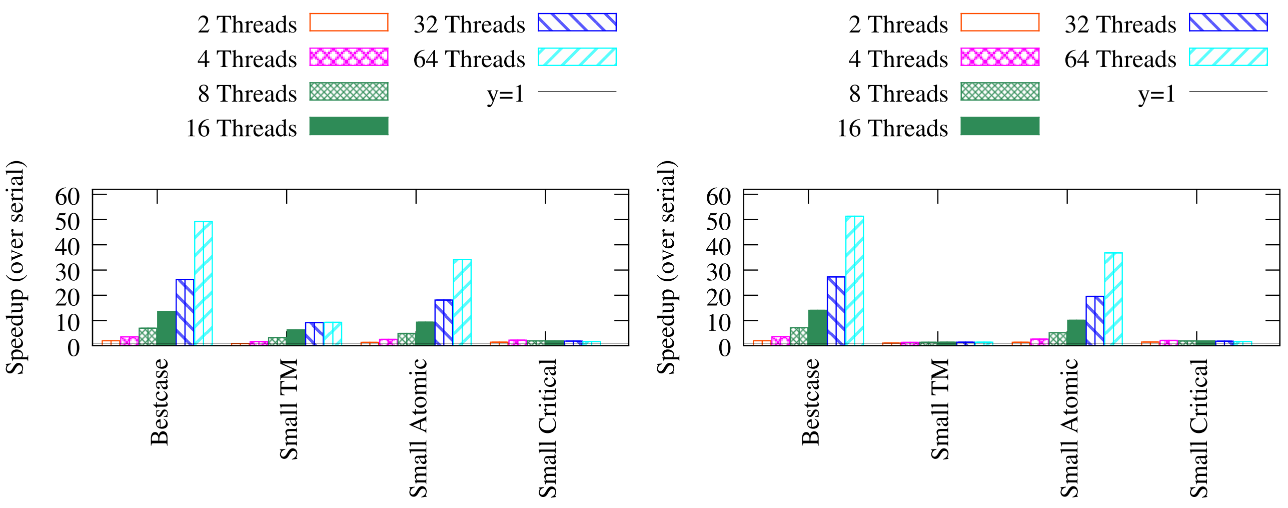


Figure 4: CLOMP-TM performing 8 divide operations per zone update with huge critical sections. Speedup shown with *None*.

`OMP_WAIT_POLICY` has been set to `ACTIVE` for all runs.

Figure 2 shows the speedup of the different synchronization mechanisms for updating one memory location. Due to the high contention generated by the Random and firstParts memory access pattern, *Small Atomic* is the method of choice for synchronization.

Large TM outperforms *Large Critical* as can be seen for both no and high contention cases of Figure 3. Thus, for critical sections with more than one memory update, TM is the preferred method. The *Huge TM* with 100 times the size of *Large TM* performs excellent in case of no contention (cf. Figure 4). Further experiments with higher contention cases reveal that this speedup is very fragile. These experiments demonstrate (and the TM statistics confirm) that longer transactions are more susceptible to contention.



(a) Speedup with *None*. (b) Speedup with *Random*.

Figure 2: CLOMP-TM performing 8 divide operations per zone update with small critical sections.

numParts	64
zonesPerPart	100
zoneSize	128
zone alignment	128
scatter	3
flopScale	1
timeScale	100
Zones per Part	100
Total Zones	6400
Zone Calc Stride	1
Extra Zone Calcs	8
Zone Calc Flag	-DDIVIDE_CALC
Zone Calc Formula	((1.0/(x+2.0))-0.5)

Table 3: Parameters for CLOMP-TM.

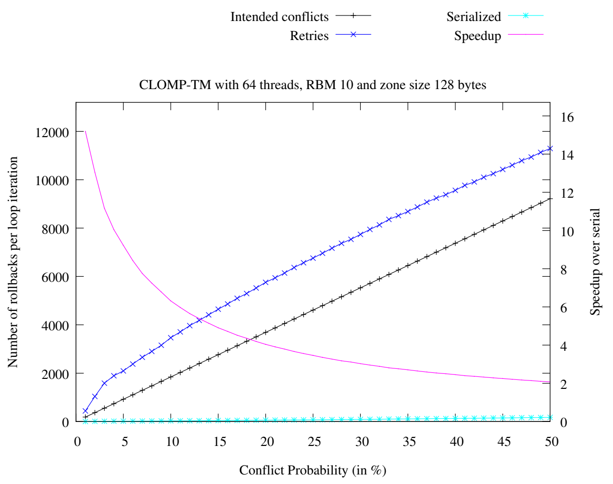


Figure 5: CLOMP-TM experiment with a zone size of 128 bytes and 64 threads. Run with `clomp-tm-bgq-divide1 -1 1 x1 d6144 128 firstZone,`*cp*`,randFirstZone 3 1 0 6 100`.

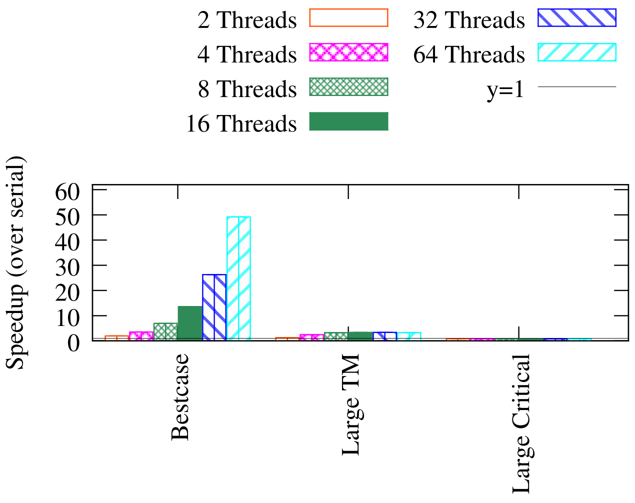
4.2 How may the memory layout affect speedup and rollbacks?

For the following experiments CLOMP-TM has been extended with a special mode that allows to transition between scatter modes. Thus, a parameter has been added that defines the number of *intended conflicts* for this run. For our experiments this parameter can be computed from a *conflict probability* (*cp*) according to the following equation: *total zones* $*$ *scatter* $*$ *cp*. Updates are counted as *intended conflicts* and performed inside a large transaction. Note, however, not all of these *intended conflicts* lead to an actual conflict and some conflicts can cause multiple rollbacks.

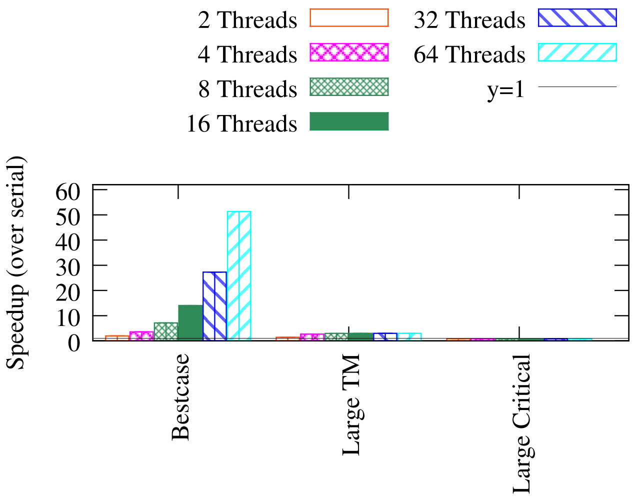
Figure 5 illustrates the impact of the number of retries on the achievable speedup. From this figure we can clearly see that a linear increase of the conflict probability leads to an exponential decrease of the speedup. In this experiment the zone size is set to 128 bytes. In case it is smaller (e.g., 64 or 32 bytes) conflicts may be falsely detected because two zones are mapped to the same cache line. These *False Positives* are eliminated when the zone size equals the size of the L2 cache line.

4.3 Performance Tuning through Environment Variables

This section studies the effect setting the environment variable `TM_MAX_NUM_ROLLBACK` (RBM) on the performance of CLOMP-TM. Figure 6 shows results with 32 threads and RBM set to 1 and 10. RBM controls the number of rollbacks before the TM run time will execute it in *irrevocable mode*. Then this transaction is executed non-transactional under a global lock so that other transactions can not interfere. In Figure 6(a) RBM is set to 10 and shows a significant higher number of retries than Figure 6(b) (RBM 1). The relative number of serialized transactions is higher for RBM=1. Both observations are due to the RBM setting because a smaller RBM value serializes after less retries. In terms of speedup RBM 10 outperforms RBM 1 because of



(a) Speedup with *None*.



(b) Speedup with *Random*.

Figure 3: CLOMP-TM performing 8 divide operations per zone update with large critical sections.

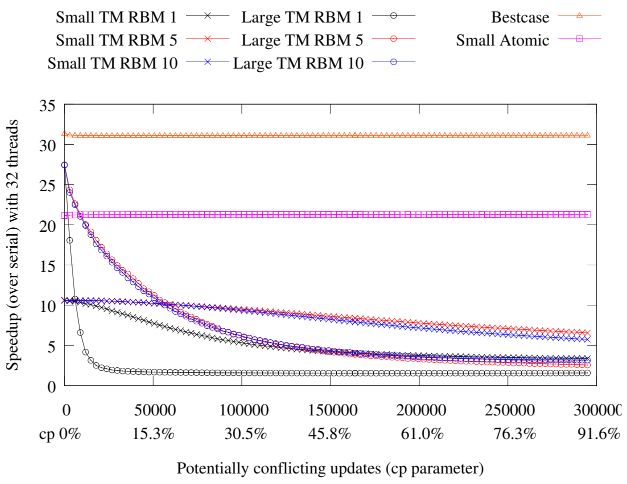


Figure 7: Studying the influence of setting `TM_MAX_NUM_ROLLBACK` **with CLOMP-TM and changing the level of contention. Run with** `clomp-tm-bgq-divide4 32 1 256 128 256 stride1,`*cp*`,stride1%/2 10 1 0 6 100`.

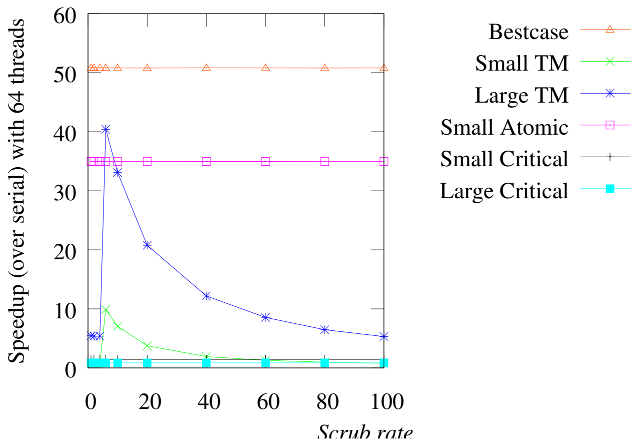


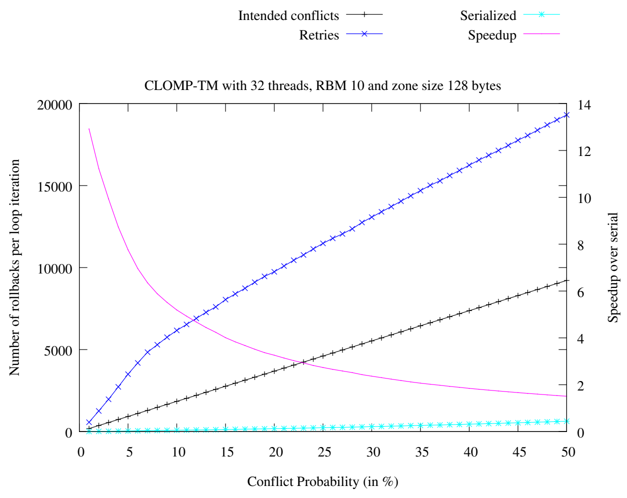
Figure 8: Influence of the scrub rate for SpecIds on performance with 64 threads. Run with `clomp-tm-bgq-divide1 -1 1 64 100 128 InPart,10,firstParts 10 1 0` *sr* `100`

the less frequent serialization. This explanation can not be generalized and needs closer investigation.

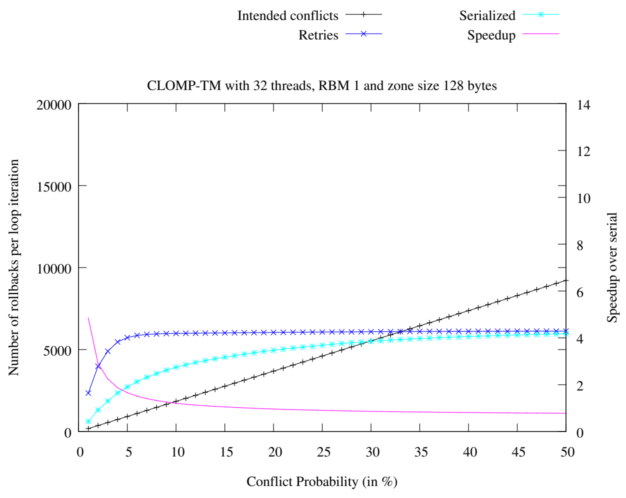
Figure 7 demonstrates the influence of RBM1, RBM5 and RBM10 on the achievable speedup with TM. For a small contention level and *Small TM*, RBM 5 shows a slightly better speedup than RBM 10. For higher contention and *Large TM* RBM 5 outperforms RBM 10 substantially. RBM 1 shows the worst performance for the presented level of contention. Setting RBM to 10 performs well in our experiments, but, depending on contention and size of a transaction, the performance of an application may increase by reducing RBM to 5.

4.4 CLOMP-TM with mixed Scatter Modes

The initial CLOMP-TM design supports only one scatter mode at a time (cf. Table 1). This leads to a fixed TM application behavior that defines the contention between threads for the whole program run. As a result, TM either performs excellent because of the lack of conflicts (e.g., scatter mode None) or suffers from the frequent retries (e.g., firstParts). We found this too restricted to model all scientific workloads and bring out the strong sides of TM. Thus, we implemented an alternating scatter mode that uses a parameter to define how often the second scatter mode will be used for updates. Increasing this parameter leads to more updates with the second scatter mode. An important parameter for TM is the scrub rate. It triggers a garbage collection for TM SpecIds. SpecIds mark entries in the cache as belonging to the same or different transactions. Figure 8 shows that varying the scrub rate has a large impact. For our benchmark with a lot of transactions and short intervals between these, a scrub rate of 6 shows the best performance.



(a) 32 threads with RBM 10.



(b) 32 threads with RBM 1.

Figure 6: Influence of `TM_MAX_NUM_ROLLBACK` (RBM) on retries/speedup. Run with `clomp-tm-bgq-divide1 -1 1 x1 d6144 128 firstZone,`*cp*`,randFirstZone 3 1 0 6 100`.

4.5 CLOMP-TM with MPI

Results with CLOMP-TM presented earlier in this paper focused on parallelization with OpenMP only. Given the size of the planned BG/Q system, scientific applications will require parallelization with MPI in order to exploit its compute power. This very same MPI parallelization can also be used to run multiple MPI tasks on one node. In the following we study the side effects of running multiple MPI tasks, each executing CLOMP-TM, on one node. Our goals are:

1. to verify the robustness of the previously presented results,
2. identify bottlenecks due to the sharing of architectural resources,
3. and determine a fitting MPI task to OpenMP thread ratio.

Similar to the previously published CLOMPI [2], we enhanced CLOMP-TM with MPI calls. Besides calls to init and finalize MPI, we inserted *MPI_Barrier*s. These barriers are placed such that all MPI tasks execute the code for the same synchronization primitive. An example for the placement of the *MPI_Barrier*s is shown in Listing 1. To execute in this lock step fashion guarantees that all MPI tasks execute the code for the same synchronization primitive. As a consequence, the contention for architectural resources that are necessary for synchronization (such as the L2 cache) is increased artificially. Thus, the methodology stresses the architectural resources needed by that synchronization primitive and will uncover bottlenecks in the implementation.

```
MPI_Barrier(MPI_COMM_WORLD);
get_timestamp (&bestcase_start_ts);
do_bestcase_version();
get_timestamp (&bestcase_end_ts);
MPI_Barrier(MPI_COMM_WORLD);
```

Listing 1: Use of MPI barriers for CLOMP-TM with MPI.

Figure 9 illustrates the performance characteristics of CLOMP-TM with MPI and small as well as large critical sections. The experiments are carried out as *strong scaling* experiments, i.e., the amount of work is constant for all task counts. We achieve this by dividing the number of parts (initially 1024) by the number of MPI tasks. The number of updates in the second scatter mode is also divided by the number of MPI tasks. All MPI tasks execute as many threads as possible without oversubscribing the node (e.g., 1 MPI task executes 64 threads).

First, Figure 9(a) shows the average speedup of the threads in each MPI task multiplied by the number of MPI tasks on the y-axis. The number of MPI tasks is plotted on the x-axis. In this case with extremely low contention, *Large TM* performs almost as well as *Small Atomic*. The surprise is that for large task counts, *Small* and *Large Critical* perform better than *Small TM*. Especially *Small Critical*, that is protecting one memory location, has been optimized heavily in the new BG/Q L2 cache and is now a strong contender for TM. This first impression is confirmed by Figure 9(b) where *Small Critical* outperforms *Small* and *Large TM* in a case with high contention. *Large TM* is still outperforming *Large Critical*, especially on small task counts. *Large Critical* benefits from the smaller thread numbers at higher task counts because the cost for serialization is reduced.

Finding a competitive task to thread ratio.

Looking at the peak performance of one BG/Q node, we see that there are 16 cores each equipped with 4-way hyper-threading. Thus, the theoretical peak performance in terms of speedup is 64. As demonstrated in earlier papers [2], an OpenMP barrier has a higher overhead for higher thread counts. Thus, a hybrid parallelization with MPI and OpenMP may achieve a higher score than the current OpenMP only implementation. In order to be able to compare results of OpenMP and hybrid parallelization, we use a simple metric. For the hybrid case, we multiply the reported OpenMP speedup with the number of MPI tasks. Figure 9 shows that the *Bestcase* across MPI tasks is stable. Depend-

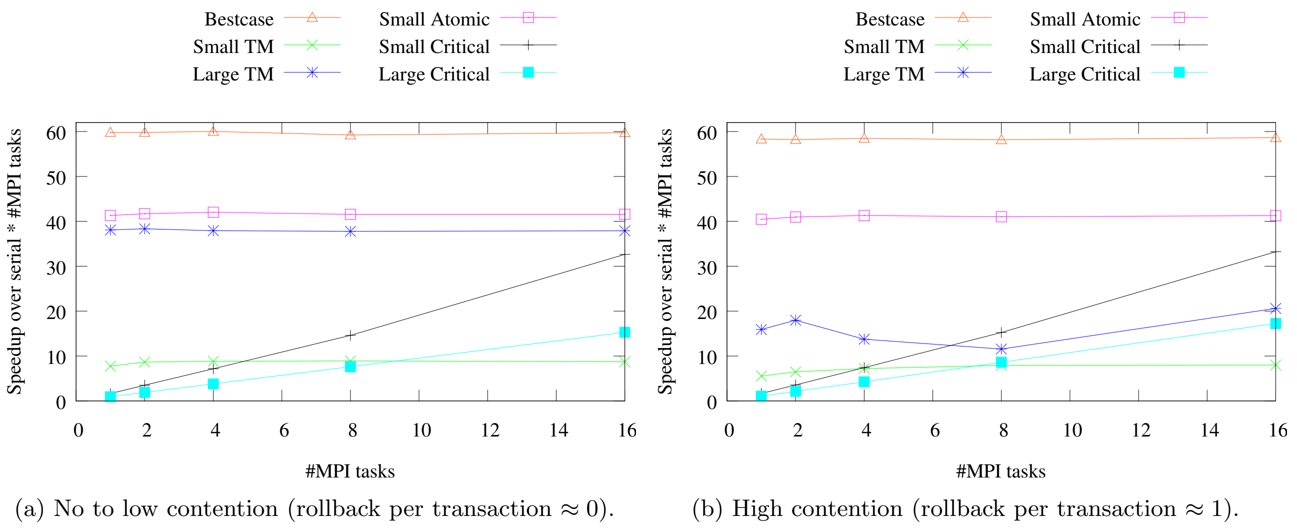


Figure 9: CLOMP-TM with MPI performing 8 divide operations with a stride of 4 per zone update with no and high contention. Run with `clomp-tm-mpi-bgq-divide4 -1 1 (1024/taskno) 128 256 stride1,`*cp*`,stride1%/2 10 1 0 6 100`. *cp* is set to $\frac{16}{taskno}$ for the left and $\frac{1.12*10^6}{taskno}$ for the figure on the right.

ing on the properties of the application the best synchronization primitive varies. Across all tested memory access patterns and MPI tasks configurations, the OpenMP version with the highest possible thread count performs best. This is an important first insight that we gain from this experiment. For architectures with hyper-threading, the additional HW threads are often turned off because they lead to a slowdown. For the BG/Q architecture running the CLOMP-TM (with MPI) benchmark this is not the case. Every thread contributes an important part of the reported performance. For the executed strong scaling experiments the results of finding a preferable task to thread ration are inconclusive. All tested ratio perform well and differences are extremely small.

The synchronization primitive with the best performance varies. In all high contention cases *Small Atomic* performs best. For cases with little to no contention *Large TM* may perform almost as good as *Small Atomic*. The large transactions benefit from the optimistic concurrency. The overhead for setting up the transaction amortizes due to the long transaction size. Unfortunately, this effect is limited to scenarios where expensive roll back operations are infrequent.

5. LESSONS LEARNED

The findings from the previous section are summarized (by listing desired code properties for TM) and condensed into best practices afterwards. The identified preferable code properties for TM are:

- critical section should have low contention so that conflicts are unlikely,
- critical sections should access more than one memory location (preferably in the range of 10 to 20) so that `omp atomic` is not applicable and TM's property of providing atomicity for updates of multiple memory locations is valuable,
- high computation to synchronization ratio so that computation can mask the overheads of synchronization.

For synchronization with OpenMP, both the size of the code region that needs to be executed atomically and the potential conflict rate play an important role:

- For code regions that only require atomic updates using one instruction, *omp atomic* shows the best performance, since it can be mapped to the efficient atomic instructions implemented in the BG/Q L2 cache.
- For larger critical sections with low to moderate contention and conflict potential (<< 1 rollback per transaction), TM using the *tm_atomic* primitive is beneficial, since any transaction conflict is amortized by avoiding serialization.
- In case of very high contention (> 1 rollback per transaction) and small critical sections, *omp critical* also outperforms TM, since the TM conflicts start dominating leading to higher overhead.
- For large critical sections and high contention (≈ 1 rollback per transaction) setting `TM_MAX_NUM_ROLLBACK` to 5 yields better results because the retrying transactions are serialized earlier, wasting less work.
- For applications that are not utilizing the full memory bandwidth with a high transactional execution time and short times in between transactions, setting the scrub rate to 6 yields better performance.

These findings complement a previous study on using Software Transactional Memory for scientific codes using a different and more specific setup [1]. Additionally, researchers already identified codes that match the criteria from above and are expected to benefit from TM [23], although also this work was limited to STM methods and has up to now not been verified on a HTM system, and thus no performance results have been published either. Our current recommendations verify the applicability of these previous preliminary studies to HTM, extend them by adding tradeoffs offered by the new performance knobs found in IBM's HTM solution, and generalize them to a more comprehensive guide for application developers.

6. TRANSACTIFYING AND OPTIMIZING MCB

In this section we apply the best practices from the previous section to a benchmark closely representing a real world application. The Monte Carlo Benchmark (MCB) models a Monte Carlo simulation. Characteristic for the Monte Carlo simulation is that an exact result is not computed directly. Instead the simulation has a probabilistic parameter and simulations are repeated until the probability of a result can be quantified. This technique is called random sampling. These Monte Carlo simulations are very popular in the context of physics simulations.

MCB is already parallelized with MPI and OpenMP. The original version uses *omp critical* and *omp atomic* to synchronize OpenMP threads. It is called *Critical & Atomic* in the following. As a first simple TM implementation, referred to as *TM simple*, all critical sections are replaced with transactions. Environment variables are set to the default for *TM simple* and *Critical & Atomic*. We apply the lessons learned from the previous Section 5 and construct a transactional version, called *TM opt*.

In *TM opt* synchronization that involves only one instruction uses *omp atomic*. Further, all *omp critical* constructs are replaced with *tm_atomic*. A profiling run with TM statistics reveals that the remaining transactions are rolling back frequently. Thus, we set `TM_MAX_NUM_ROLLBACK` to 5 for *TM opt* to speedup serialization. This reduces the amount of wasted work and has been shown to yield speedups in Section 4.3.

Table 6 holds the results of the experiments with one MPI task and 64 threads in a strong scaling experiment with $5 * 10^6$ particles. Each value is an average of tracks per second over 30 runs and normalized to baseline: tracks per second with one MPI task and one thread. *TM opt* performs very well with a speedup over baseline with 26.47. The result of *TM simple* demonstrates that the lessons learned in this paper are viable for programming with TM. Further experiments reveal a limited potential for optimizing the synchronization of threads in MCB. Commenting out all occurences of `omp atomic` and `omp critical` (and getting the wrong answer for the simulation) yields $\approx 5\%$ perfomance improvement. This finding makes the achieved score of *TM opt* even more precious.

Code version	*Critical & Atomic*	*TM simple*	*TM opt*
Speedup	25.80	21.50	26.47

Table 4: MCB with one MPI tasks and 64 threads (strongScaling) – speedup over baseline.

7. CONCLUSIONS

In this paper we evaluated BG/Q's TM hardware from the perspective of an application developer. We introduced, CLOMP-TM, a benchmark designed to represent scientific applications, and applied it to benchmark transactions against traditional synchronization primitives, such as *omp atomic* and *omp critical*. We then extended CLOMP-TM with MPI to mimic hybrid parallelization with OpenMP and MPI. Additionally, we studied the impact of environment variables on the performance. Finally, we condensed the findings into a set of best practices and applied them to a Monte Carlo Benchmark that closely resembles real world applications. An optimized TM version of MCB with 64 threads achieved a speedup of 26.47 over the baseline.

8. REFERENCES

[1] B. L. Bihari. Applicability of transactional memory to modern codes. In *International Conference on Numerical Analysis and Applied Mathematics 2010 (ICNAAM 2010) Conference Proceedings*, pages 1764–1767, Rodos, Greece, 2010. APS.

[2] G. Bronevetsky, J. Gyllenhaal, and B. R. De Supinski. Clomp: accurately characterizing openmp application overheads. In *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, IWOMP'08, pages 13–25, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[4] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[5] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance sparc cmt processor. *Micro, IEEE*, 29(2):6 –16, march-april 2009.

[6] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière. Evaluation of amd's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 27–40, New York, NY, USA, 2010. ACM.

[7] C. Click. Azul's experiences with hardware transactional memory, Jan 2009. In HP Labs - Bay Area Workshop on Transactional Memory.

[8] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 336–346, New York, NY, USA, 2006. ACM.

[9] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. *SIGPLAN Not.*, 44:157–168, Mar. 2009.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[11] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of*

parallel programming, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.

[12] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. 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] R. Haring. The Blue Gene/Q Compute Chip. In *Hot Chips 23*, August 2011.

[14] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*, volume 5. Morgan & Claypool Publishers, 2010. 2nd edition, Synthesis Lectures on Computer Architecture.

[15] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[16] C. Kachris and C. Kulkarni. Configurable transactional memory. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 65–72, Washington, DC, USA, 2007. IEEE Computer Society.

[17] S. Kang and D. A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In *PPoPP '09: Proc. 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 15–24, feb 2009.

[18] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT'09: Workshop on Transactional Computing*, February 2009.

[19] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, aug 2007.

[20] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun. Atlas: A chip-multiprocessor with transactional memory support. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1 –6, april 2007.

[21] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[22] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[23] M. Wong, B. L. Bihari, B. R. de Supinski, P. Wu, M. M. amd Y. Liu, and W. Chen. A case for including transactions in OpenMP. In *IWOMP 2010 Conference Proceedings*, pages 149–160, Tsukuba, Japan, June 2010. LNCS 6132.

Prepared by LLNL under Contract DE-AC52-07NA27344