Transactional Event Profiling in a Best-Effort Hardware Transactional Memory System

Matthew Gaudet Advisor: José Nelson Amaral
Dept. of Computer Science, University of Alberta
Edmonton, Alberta, Canada
mgaudet@ualberta.ca

ABSTRACT

Blue Gene/Q's (BG/Q) unique transactional memory system provides hardware isolation, atomicity and consistency for memory locations while leaving the details of the transactional programming system to software layers above the hardware [22]. This design allows for complex systems implemented as part of the software runtime. Here a profiling extension to the software runtime is presented, which allows for in-depth analysis of the actions of the transactional memory runtime system, as well as giving insight into the behaviour of the program being profiled.

1. INTRODUCTION

As hardware becomes more parallel due to the demise of frequency scaling, and the increasing importance of energy efficiency, parallel programming has become a topic of great interest.

One of the pitfalls in parallel computation are *race conditions*, which are cases where the result of a computation can change depending on the order in which two threads performed some operations. This can happen when two threads computation on shared data is interleaved. Traditionally the solution to race conditions is synchronization, using structures such as locks to delineate critical sections into which only one thread may proceed at once. This kind of synchronization is *pessimistic*, in that even if the operations may not cause a problem, mutual exclusion will nevertheless be enforced. This can reduce an application's parallelism if the waiting on locks becomes a bottleneck.

Furthermore, programming with locks can become difficult because as soon as there is more than two locks, the possibility of *deadlock* occurs, wherein two threads each hold a lock the other needs, and so no further progress is possible. The difficulty of programming with locks becomes further exacerbated with libraries and callbacks involved, as deadlocks can occur with no clear programmer error [9].

The difficulties of traditional parallel programming, combined with the requirement for more parallelism, have made programming models which make parallel programming easier of great interest. One such model, Transactional Memory (TM), has received large amounts of attention.

Transactional Memory provides programmers with the notion of an atomic region, which can be any number of statements in a program which must appear to occur atomically, without interference from other threads, or not at all. In practice, TM is an *optimistic* concurrency construct, as it lets threads enter a atomic section in parallel, but monitors the actions of the threads, aborting and retrying transactions which will violate the atomicity property.

TM has received much attention due to the performance possibilities inherent in its optimistic concurrency model, and its simple composable semantics which disallow deadlock.

TM was initially specified as a hardware feature by Herlihy and Moss [12]. However, for much of its history it was mostly implemented as Software Transactional Memory (STM), where reads and writes to shared data are annotated with calls to a runtime library, either by the user or by the compiler [7, 8, 5].

Hardware Transactional Memory (HTM) designs have also existed as simulations, with a number of simulators used in academic studies [17, 21]. Recently however HTM designs have begun to make their way from simulations to actual silicon [6, 4, 22, 14, 13].

2. PROBLEM AND MOTIVATION

In Transactional Memory systems, conflict resolution refers to the choice of which transaction will abort when two transactions conflict. In STM systems this decision is often made through a module called a contention manager, which takes into account various factors to decide what is the most advantageous transaction to abort [11, 18, 19].

In BG/Q there is not as much control, as it is always the younger transaction that aborts. This is decided by the kernel. Once a rollback is initiated after abortion, the BG/Q runtime system becomes involved and makes a decision to either retry the aborted transaction speculatively, or to execute an operation called *serialization*. A serialized transaction is forced to completion on retry, possibly at the expense of otherwise non-conflicting transactions, which will abort themselves if they conflict or attempt to commit.

A serialized transaction runs without speculation, which guarantees forward progress in the case of repeated failures, such as those caused by insufficient storage for speculative state. Serializing too little can cause persistent failures to repeat, wasting resources and slowing program execution. Serializing too often can abort transactions that otherwise should have committed, also wasting resources and slowing program execution. Static serialization decisions are inappropriate because the correct choice of when to serialize can

Copyright is held by the author/owner(s).
ACM X-XXXXX-XX-X/XX/XX.

vary depending on input data and on other transactions. We are investigating runtime adaptation heuristics to make dynamic serialization decisions.

Many TM implementations, including BG/Q, provide summary counters of the number of committed and aborted transactions. However, for the design of an adaptation heuristic, these summaries are insufficient.

For example, consider two transactional workloads each of which incurs 10,000 aborts, where one takes 1 second to run, and the other takes 10 seconds. Clearly these two programs experience very different contention levels, yet this remains invisible in summary counters.

Disambiguating these kinds of confusing situations, which arise often when designing adaptation heuristics, was the main motivation behind the design of our Event Profiler which gives us insight into dynamic conditions by capturing the time-series of transactional events inside the TM system.

3. BACKGROUND AND RELATED WORK

3.1 BG/Q's TM Architecture

Blue Gene/Q is unique in having a three layer architecture for Transactional memory programs, as shown in Figure 1. Most other proposed HTM systems instead directly connect user programs to the hardware, exposing TM through the instruction set architecture.

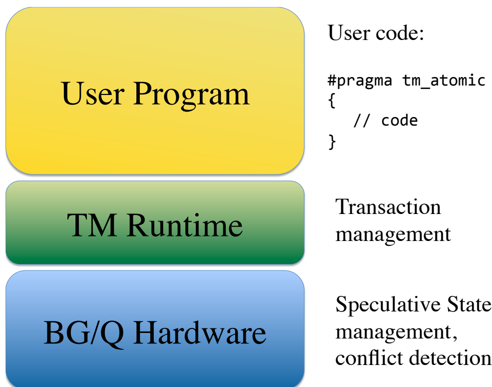

Figure 1: BG/Q TM architecture.

A user of BG/Q's TM system defines atomic regions in their code. These atomic regions are compiled into calls to the TM runtime by the compiler.

The TM runtime is responsible for setting up the transaction before running the user code, managing aborted transactions, and executing the transaction commit operations at transaction end.

Transaction setup involves saving a minimal amount of restore information, as well as starting speculation on the processor. Transaction aborts are managed by choosing to either retry the transaction, or to *serialize*, which runs transaction code non-transactionally, while disallowing interference from other transactions. On transaction end, the runtime asks the processor to commit the data, and executes the abort code if this fails.

During each step of this process some summary statistic counters are gathered which provide some insight into the progress of a running program.

The BG/Q TM hardware is implemented in the L2 cache, and allows up to 20 MB of speculative state to be stored. One side effect of this design is the question of how each of the L1 caches on the individual A2 cores are handled with regards to TM. There are two different modes, known as Short-Running (SR) mode, and Long-Running (LR) modes. Each is optimized for a different transaction characteristic, and has different performance tradeoffs. SR mode is faster to start, however penalizes reads to speculatively modified cache lines, forcing requests for these lines to bypass the L1 cache. LR mode is slower to start, as it flushes the entire L1, but does not have the same read-after-write penalty. We analyze these two modes with our profiler on the application `genome` in Section 5.1.1.

3.2 Related Work

Time-series analysis of transactional memory programs has been rare. Previous work has been done on Software Transactional Memory Systems, or has required specialized hardware to be added to the processors [23, 1, 3]. The existence of the TM runtime in BG/Q's HTM makes inserting the event profiling trivial as compared to other HTM systems where the TM code is tightly woven with application code.

Lev created a prototype transactional profiler called *T-PASS*[1] which used stub calls read by the DTrace dynamic tracing framework to implement the profiling. Lev's profiler could profile at a deeper level than the one presented here, however, those features relied on it having been built specifically for the SkySTM runtime [15].

Zyulkyarov describe a transactional profiler for an STM augmented C# system, and described a visualization technique which inspired our visualization in Figure 4, though they did not explore the possible insight of aggregate data visualization [23].

TMProf is a transactional memory profiler intended for user usage [10]. They create application traces similar to ours, however use an interactive graphical front-end in order to display a micro-level analysis of the transactional execution in order to visually indicate areas of possible improvement in the algorithms. The focus of *TMProf* is on micro-level interactions and so they don't discuss any aggregate analyses.

Manual tuning of TM applications often requires an in depth understanding of the application, which could benefit from detailed profiles like those presented here [20].

4. APPROACH AND UNIQUENESS

4.1 Event Profiling

The approach presented here is a software profiling system that provides a high-resolution time-series view of transactional behaviour in programs run on BG/Q. This system provides both macro and micro views of transactional behaviour that have influenced our understanding of adaptation and subsumed simpler summary statistics.

In this profiler, event profiles are gathered by the TM runtime on events such as begin, rollback, serialization and commit. These events are stored in per-thread buffers, and are dumped to a log-file at the end of program execution by the runtime's `atexit` shutdown handler. An event profile con-

[1]Transactional Program Analysis System

sists of a list of event tuples: (TS,T,TX,O,[A_1,A_1]), where: TS is the timestamp read from a high-resolution counter, T is an event type, TX is a unique identifier for each static transaction and O is the originating logical thread ID. A_1 and A_2 are optional auxiliary data which can be used to save other "extra" data to the profile. For example, this could be used to save the number of rollbacks used on each abort.

The raw event profiles are post-processed by analyzers, which are discussed in more detail in section 5. The loose coupling between the profiles and the analyzers is one of the unique features of this system.

4.1.1 Limitations

Some proposed transactional profiling systems not only track begin and end events, but also the read and write sets [15]. While this can provide interesting data on conflicting data, it is impossible in the current form of BG/Q's TM system since read-write sets are managed in the hardware, in such a way that they are not exposed to the software system.

Event profiles are imprecise because probe effects may change the behaviour under study. This imprecision is acceptable for analyses that use aggregates, however must be kept in mind when doing micro-level analysis.

Probe Effect refers to the disruption of the original programs running when instrumented by our profiler. Probe effects are impossible to avoid completely, however we made design choices to minimize them to the extent possible.

First, the event profile buffers are thread local which avoids any synchronization overhead for the profiling. Second, the event profiling functions are made as short as possible, and inlined to avoid function call overhead inside the runtime handlers. Third, the event profiling system avoids memory allocation as much as possible by allocating buffers in very large chunks.

A fourth, space saving measure, is to not explicitly store the thread origin in either the in-memory buffer or on-disk representation. This is done by outputting the buffers in thread order. Analyzers can infer the thread count by watching for breaks in timestamp monotonicity as they process the profiles, which indicates a thread change.

Table 1 shows the average overhead computed on the average runtime of 3 different runs on a quiescent machine for 1-64 threads. Negative numbers indicate the program ran faster with profiling active. This can happen in transactional programs because event profiling can change the transaction ordering to be more beneficial, spacing out transactions to reduce the number or rate of aborts. This can be seen empirically in Table 2, which shows how the rollback count is affected by enabling event profiling.

`genome` spends relatively little time overall in transactions however, and so is a lower bound on the possible impact of the event profiling. Benchmarks which spend a much larger ratio of their time inside transactions will spend a correspondingly larger ratio of time inside profiling code, and will thus have a larger probe effect, as seen in the `kmeans` columns of Tables 1 and 2

Table 1: Profiling overhead in `genome` and `kmeans` (% of profile-free running time)

	`genome`		`kmeans`	
Threads	SR Mode	LR Mode	SR Mode	LR Mode
1	0.7	0.9	4.8	3.9
2	0.9	0.6	-2.1	2.4
4	-1.1	0.8	0.5	3.3
8	-0.2	-1.3	13.0	7.8
16	-0.4	-1.7	-10.3	-11.8
32	-3.2	0.8	-23.8	-20.8
64	-5.9	0.1	0.6	-18.3

Table 2: Profiling effect on rollback counts in `genome` and `kmeans` (% of profile-free rollbacks)

	`genome`		`kmeans`	
Threads	SR Mode	LR Mode	SR Mode	LR Mode
2	-8.8	-11.6	-8.0	-2.3
4	3.2	-14.5	-4.3	-2.3
8	-4.5	-4.4	12.6	1.6
16	0.1	-50.2	-14.8	-15.2
32	-6.9	-4.6	-13.4	-2.1
64	-10.1	0.5	0.0	-3.2

5. RESULTS AND CONTRIBUTIONS

The raw event profiles can undergo a variety of analyses and visualizations, limited only by the imagination of the implementor. We have made substantial use of the event profiles, and have created a number of bespoke, one of a kind, analyzers to answer specific questions about implementations. Here we discuss analyses and visualizations we believe to be of general importance, or those which provided interesting results and lessons.

- Visualization of a time-series constructed from instantaneous estimates of events-per-second aggregated by transaction (TX) can identify phased executions. Similarly, aggregation on event type (T) reveals changes in transaction begin, commit, and rollback rates.

  Visualization of these time-series presented a challenge due to the large size of the data sets. Any visualization of reasonable size required that the data be condensed in some fashion, however, presenting overall trends without completely hiding high frequency variations is difficult. The compromise chosen here was to use aggregate the data based on fixed number of segments, but then to further aggregate the data into larger segments which are presented as box-plots to show the distribution of data over a large segment. This presentation minimizes the difficulty in choosing appropriate aggregation parameters. We have has used 10000 segments, divided into 100 box plots for each time-series presented here.

- Histograms of dynamic transaction lengths reveal the variability in the runtime of transactions and helps determine both transactions that dominate running time, and the multi-modal behaviour of some transactions which occur due to multiple control flow paths through transactions or varying amounts of contention experienced by transactions.

- Analysis of the behaviour of individual dynamic transactions allows insight into the process by which an in-

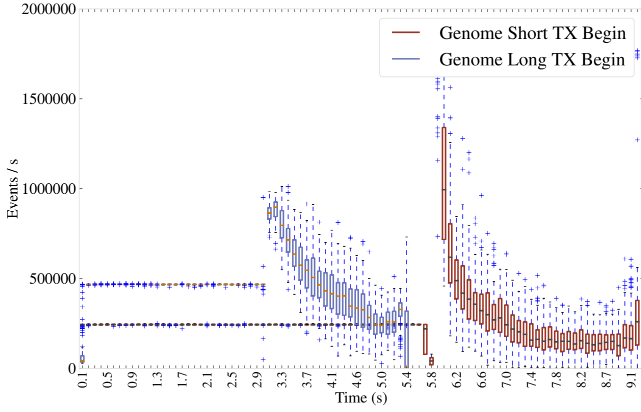

Figure 2: Transaction starts, Genome, t=16, Long Running mode vs Short Running mode.

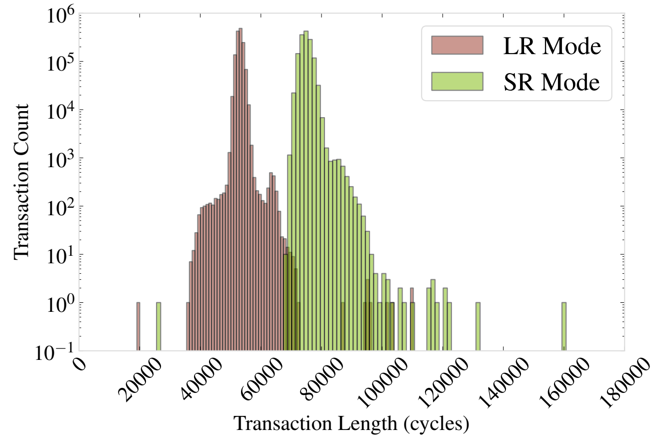

Figure 3: Comparing transaction runtime for LR and SR mode

dividual transaction is committed, and the timeline of events which lead to that point.

- Because we can annotate an event profile tuple with any value from the runtime, we have made use of the event profiles to understand the behaviour of new adaptation schemes with a variety of bespoke analyses. Visualizing parameter values as they fluctuate over time has helped discover flaws and inspired new approaches.

5.1 Selected Contributions:

5.1.1 *Short running mode vs long-running mode*

When comparing long and short execution modes in BG/Q it becomes apparent that the *rate* of transactional events is very important.

Figure 2 compares the event profiles obtained from STAMP benchmark `genome` run using long-running mode and short-running mode [16]. In many benchmarks, like `genome`, long running mode often has higher sustained rates, leading to faster execution. Furthermore, if one compares the runtime histograms for the same transaction, even with only 1 thread running, the difference in expected runtimes can be seen. This is presented in Figure 3.

One of the notable things about event rates is that they will vary substantially across a run of a program. Comparing summaries can lead to incorrect interpretations of scaling differences. This fact also provides evidence for our belief that runtime adaptation heuristics will be key to improving TM performance in TM systems.

Visualizing how adaptation changes different phases of program behaviour, by comparing event profiles of runs with adaptation enabled against those with adaptation disabled, is an important part of the design of an adaptation policy, because it can reveal pathologies otherwise invisible in summary data.

5.1.2 *Diagnosing Performance Pathologies*

Previous work has discussed a number of performance pathologies which have become part of the terminology when discussing transactional memory performance [2]. One of these pathologies is Restart-Convoy, which occurs when a group of threads all conflict with each other and restart nearly simulatanously. This leads to a group of threads aborting and restarting in waves. Often this occurs when multiple instances of the same source level transaction are in flight, because the accesses of each transaction are very similar.

By creating a visualization where individual thread's transactions are shown, we can see this situation occurring with a version of the TM runtime where we do not have any randomized backoff, as shown for the first 50,000 cycles of genome in Figure 4. Each box in the figure is a transactional event recorded in the profile. The Y axis indicates in which thread the event was recorded, and the X axis placement indicates the timestamp of the event. Retry Max events occur when a transaction exhausts its allocated number of retries and then serializes. In this particular profile, the restart convoys are being created as a side effect of all transactions being restarted after serializations caused by threads exhausting their retries (notice the blue boxes followed by green boxes leading into a restart convoy).

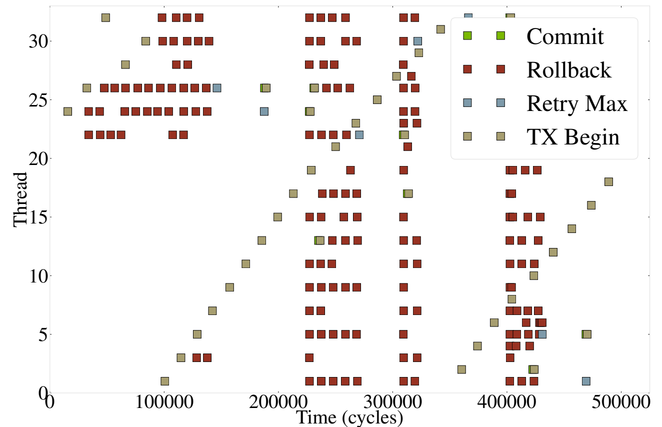

Figure 4: Restart convoys (vertical red bars) happening in `genome` run with 32 threads

6. FUTURE WORK

Currently event profiling is used only in specific analysis test on specially enabled runtimes. It is however possible to

use much of the same framework to gather online statistics in order to feed an adaptation scheme which can make use of recent history.

While we are unable to directly discover which transactions are contending with which, no attempt has yet been made to infer this kind of information through a temporal correlation study.

TProf and Zyulkyarov's transaction visualizer indicate that there is much benefit in making the visualizations interactive and zoomable. We would like to provide this kind of functionality, as well as make general performance improvements to the analyses, which are currently too slow for interactive work.

7. CONCLUSION:

We have designed and implemented a new tool for development and tuning of transactional systems. The profiles and visualizations provide a detailed view into the behaviour of TM applications, and have proved invaluable in helping understand the system and performance of applications.

Visualizations and analyses provided by this event profiler will help improve TM performance, which makes parallel programming more accessible by providing an easy to use, high performance concurrency construct.

References

[1] Ansari, M., Jarvis, K., Kotselidis, C., Luján, M., Kirkham, C., and Watson, I. Profiling Transactional Memory Applications. In *Parallel, Distributed, and Network-based Processing (PDP)* (Feb 2009), pp. 11–20.

[2] Bobba, J., Moore, K., Volos, H., Yen, L., Hill, M. D., Swift, M. M., and Wood, D. A. Performance pathologies in hardware transactional memory. In *International Conference on Computer Architecture (ISCA)* (San Diego, CA, USA, 2007), pp. 81–91.

[3] Chafi, H., Minh, C. C., McDonald, A., Carlstrom, B. D., Chung, J., Hammond, L., Kozyrakis, C., and Olukotun, K. TAPE: a Transactional Application Profiling Environment. In *International Conference on Supercomputing (ICS)* (New York, NY, USA, 2005), ICS '05, ACM, pp. 199–208.

[4] Chaudhry, S., Cypher, R., Ekman, M., Karlsson, M., Landin, A., Yip, S., Zeffer, H., and Tremblay, M. Simultaneous speculative threading: a novel pipeline architecture implemented in Sun's Rock processor. In *International Conference on Computer Architecture (ISCA)* (Austin, TX, USA, 2009), pp. 484–495.

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

[6] Click, C. Azul's experiences with hardware transactional memory. In *HP Labs' Bay Area Workshop on Transactional Memory* (2009).

[7] Felber, P., Fetzer, C., Marlier, P., and Riegel, T. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems 21*, 12 (December 2010), 1793–1807.

[8] Felber, P., Fetzer, C., and Riegel, T. Dynamic performance tuning of word-based software transactional memory. In *Principles and practice of parallel programming* (Salt Lake City, UT, USA, February 2008), pp. 237–246.

[9] Gottschlich, J. E., and Boehm, H.-J. Generic programming needs transactional memory. In *ACM SIGPLAN Workshop on Transactional computing (TRANSACT)* (March 2013).

[10] Gottschlich, J. E., Herlihy, M. P., Pokam, G. A., and Siek, J. G. Visualizing transactional memory. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2012), PACT '12, ACM, pp. 159–170.

[11] Herlihy, M., Luchangco, V., Moir, M., and William N. Scherer, I. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proc. 22nd ACM Symposium on Principles of Distributed Computing* (Jul 2003), pp. 92–101.

[12] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture (ISCA)* (1993), pp. 289–300.

[13] Jacobi, C., Slegel, T. J., and Greiner, D. F. Transactional memory architecture and implementation for ibm system z. In *MICRO* (2012), IEEE Computer Society, pp. 25–36.

[14] Kanter, D. Analysis of Haswell's transactional memory. http://www.realworldtech.com/page.cfm?ArticleID=RWT021512050738&p=1, September 2012. Real World Technologies.

[15] Lev, Y. *Debugging and Profiling of Transactional Programs*. PhD thesis, Brown University, 2010.

[16] Minh, C. C., Chung, J., C.Kozyrakis, and Olukotun, K. Stamp: Stanford transactional applications for multi-processing. In *International Symposium on Workload Characterization (IISWC)* (Seattle, WA, USA, September 2008), pp. 35–46.

[17] Moore, K., Bobba, J., Moravan, M., Hill, M., and Wood, D. Logtm: log-based transactional memory. In *High-Performance Computer Architecture (HPCA)* (2006), pp. 254–265.

[18] Scherer III, W. N., and Scott, M. L. Contention management in dynamic software transactional memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs* (St. John's, NL, Canada, Jul 2004).

[19] Scherer III, W. N., and Scott, M. L. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing* (Las Vegas, NV, Jul 2005).

[20] Schindewolf, M., Bihari, B., Gyllenhaal, J., Schulz, M., Wang, A., and Karl, W. What scientific applications can benefit from hardware

transactional memory? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 90:1–90:11.

[21] Shriraman, A., Dwarkadas, S., and Scott, M. L. Flexible decoupled transactional memory support. In *International Conference on Computer Architecture (ISCA)* (Washington, DC, USA, 2008), pp. 139–150.

[22] Wang, A., Gaudet, M., Wu, P., Ohmacht, M., Amaral, J. N., Barton, C., Silvera, R., and Michael, M. M. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Parallel Architectures and Compilation Techniques (PACT)* (Sept 2012).

[23] Zyulkyarov, F., Stipic, S., Unsal, O. S., Cristal, A., Harris, T., Hur, I., and Valero, M. Profiling and Optimizing Transactional Memory Applications. *International Journal of Parallel Programming 40* (February 2012), 25–56.