Advanced Concurrency Control for Transactional Memory using Transaction Commit Rate

Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson

The University of Manchester
{ansari,kotselidis,jarvis,mikel,chris,watson}@cs.manchester.ac.uk

Abstract. Concurrency control for Transactional Memory (TM) is investigated as a means for improving resource usage by adjusting dynamically the number of threads concurrently executing transactions. The proposed control system takes as feedback the measured *Transaction Commit Rate* to adjust the concurrency. Through an extensive evaluation, a new Concurrency Control Algorithm (CCA), called P-only Concurrency Control (PoCC), is shown to perform better than our other four proposed CCAs for a synthetic benchmark, and the STAMP and Lee-TM benchmarks.

1 Introduction

Explicit concurrent programming using fine-grain locks is known to be challenging for developing robust and correct applications. However, the need to simplify concurrent programming has become a priority with the prospect of concurrent programming becoming mainstream to take advantage of multi-core processors. Transactional Memory (TM) [1, 2] is a new concurrent programming paradigm that aims to ease the complexity of concurrent programming, yet still offer performance and scalability competitive with fine-grain locking.

TM requires developers to mark code blocks that access shared data structures as *transactions*. A runtime layer manages transactions' concurrent data accesses. Conflicts between any two transactions occur when one attempts to modify data previously modified or read by another active transaction. This conflict for shared data is resolved by a *contention management policy* [3] that decides to *abort* one transaction, and let the other continue. A transaction *commits* if it executes its code block without being aborted, making globally visible its modifications to shared data.

The motivation for this work is the observation that TM applications exhibit fluctuating amounts of *exploitable parallelism*, i.e. the maximum number of transactions that can be committed concurrently, without any aborts, varies over time. We hypothesize that dynamically adjusting the number of transactions allowed to execute concurrently in response to the fluctuating exploitable parallelism should improve resource usage when exploitable parallelism is low,

and improve execution time when it is high. *Transaction Commit Rate* (TCR), the percentage of committed transactions out of all executed transactions in a sample period, is investigated as a suitable application-independent measure of exploitable parallelism.

The only previous work is our own proposal of four Concurrency Control Algorithms (CCAs) [4]. This paper evaluates a new CCA, called *P-only Concurrency Controller* (PoCC), against the previously proposed CCAs using a synthetic benchmark and the STAMP and Lee-TM benchmarks. The new CCA is called *P-only Concurrency Controller* (PoCC). The evaluation explores the effect of the CCAs on the benchmarks' execution time, resource usage, wasted work, aborts per commit (APC), and responsiveness to changes in TCR. The results show PoCC gives similar or better performance, is more responsive to TCR changes, and more robust to noise in TCR changes, than the previous CCAs.

Section 2 introduces PoCC, and compares it to the other four CCAs. Section 3 details the experimental platform used in the evaluation, and Section 4 presents the results. Finally, Section 5 summarizes the paper.

2 P-only Concurrency Control

Using control theory terminology, the control objective is to maintain the *process variable* TCR at a *set point* desirable value, in spite of *unmeasured disturbance* from fluctuating exploitable parallelism. The *controller output* is to modify the number of transactions executing concurrently in response to changes in TCR. For the purposes of this paper, the number of transactions executed concurrently is controlled by enabling or disabling threads that execute transactions. PoCC is based on a P-only controller [5] and operates as a loop:

1. If `currentTime` − `lastSampleTime` < *samplePeriod*, goto Step 1;
2. If `numTransactions` < *minTransactions*, goto Step 1;
3. `TCR` ← `numCommits` / `numTransactions` × 100;
4. Δ`TCR` ← `TCR` − *setPoint*;
5. If (`numCurrentThreads`= 1) & (`TCR` > *setPoint*);
   (a) then Δ`threads` ← 1;
   (b) else Δ`threads` ← Δ`TCR` × `numCurrentThreads` / 100 (rounded to the closest integer);
6. `newThreads` ← `numCurrentThreads` + Δ`threads`;
7. Adjust *minThreads* ≤ `newThreads` ≤ *maxThreads*;
8. `numCurrentThreads` ← `newThreads`;
9. Set `lastSampleTime` ← `currentTime`, go to Step 1;

where the parameters are:

- *samplePeriod* is the sample period (tunable);
- *minTransactions* is the minimum number of transactions required in a sample (tunable);
- *setPoint* is the set point value (tunable);

- *minThreads* is the minimum number of threads, for this paper one; and
- *maxThreads* is the maximum number of threads, for this paper the maximum number of processors, or cores, available.

The *setPoint* determines how conservative PoCC is towards resource usage efficiency. A high *setPoint*, e.g. 90%, is quick to reduce threads when TCR decreases, but slow to adapt to a sudden large increase in TCR, and vice versa. The evaluation in Section 4 shows that maintaining a fairly high *setPoint* of 70% does not result in performance degradation.

PoCC calculates Δ`threads` using the relative gain formula described in Step 5, which allows *setPoint* to be a value, rather than a range. The other four CCAs (see [4] for their description) use a range of 50-80% to calculate Δ`threads` with an absolute gain formula, and, thus, a small Δ`TCR` leads for certain to a modified `numCurrentThreads`. In addition over large thread counts, the range produces a coarse-grain control. In contrast, PoCC, using the relative gain formula (Step 5), allows Δ`threads` to be zero at low thread counts in response to small Δ`TCR`, and allows also fine-grain control at large thread counts.

Compared with the other four CCAs, PoCC adds a new parameter, *minTransactions*, that acts as a filter against noisy TCR profiles such as in Figure 6. Such noisy samples may occur due to the average transaction execution time being longer than the *samplePeriod*. Few transactions execute every *samplePeriod*, and thus their outcomes (abort/commit) heavily bias TCR. The other CCAs, lacking PoCC's filter, absorbed noise by using a large *samplePeriod*, which is a trade-off with responsiveness, as shown in Section 4.4. However, this may mean that *samplePeriod* needs to be re-tuned for each new application to avoid either slow or noisy responses. In PoCC, *samplePeriod* is determined based on the overhead of executing the control loop. This reduces the application dependence, and makes PoCC suitable for general use.

3 Experimental Platform

3.1 Concurrency Control Parameters

PoCC is implemented in DSTM2, a state-of-the-art Java-based software TM implementation [6]. PoCC's parameters are: *setPoint* is 70%, *minTransactions* is 100. Experimental evaluation found PoCC took on average 2ms to execute its loop, thus *samplePeriod* is set to 1 second to make its overhead negligible.

The other four CCAs (see [4] for more details) are also evaluated, and abbreviated as SA (SimpleAdjust), EI (ExponentialInterval), EA (ExponentialAdjust), and EC (ExponentialCombined). Their configuration is left to their default values.

3.2 Software & Hardware Platform

All popular contention managers [7–9] have been used, but only results for the Priority contention manager are presented, as it gives the best execution times

when executing without concurrency control. The Priority manager prioritizes transactions by start time, aborting younger transactions on conflict. The platform used for the evaluation is a 4x dual core (8 core) AMD Opteron 2.4GHz system with 16GB RAM, openSUSE 10.1, and Java 1.6 64-bit using the parameters `-Xms1024m -Xmx14000m`. As the system has a maximum of 8 cores, all benchmarks are executed using 1, 2, 4, and 8 initial threads (except for the synthetic benchmark, see below). We use the term initial threads as concurrency controlled execution may dynamically change at runtime the number of threads (between 1 and 8).

3.3 Benchmarks

One synthetic and seven real, complex benchmark configurations are used. The synthetic benchmark and each complex benchmark configuration is executed five times, and the results averaged. The synthetic benchmark, StepChange, oscillates the TCR from 80% to 20% in steps of 20% every 20 seconds (as seen in Figure 6), and executes for a fixed 300 seconds. StepChange needs to be executed with the maximum 8 threads to allow its TCR oscillation to have impact, as it operates by controlling the number of threads executing committed or aborted transactions.

Configuration Name	Application	Configuration
StepChange	StepChange	max_tcr:80, min_tcr:20, time:300, step_size:20, step_period:20,
Genome	Genome	gene_length:16384, segment_length:64, num_segments:4194304
KMeansL	KMeans low contention	clusters:40, threshold:0.00001, input_file:random10000_12
KMeansH	KMeans high contention	clusters:20, threshold:0.00001, input_file:random10000_12
VacL	Vacation low contention	relations:65536, percent_of_relations_queried:90, queries_per_transaction:4, number_of_transactions:4194304
VacH	Vacation high contention	relations:65536, percent_of_relations_queried:10, queries_per_transaction:8, number_of_transactions:4194304
Lee-TM-ter	Lee low contention	early_release:true, file:mainboard
Lee-TM-t	Lee high contention	early_release:false, file:mainboard

Table 1. Benchmark configuration parameters used in the evaluation.

The complex benchmarks used are Lee's routing algorithm [10], and the STAMP [11] benchmarks Genome, KMeans, and Vacation, from STAMP version

0.9.5, all ported to execute under DSTM2. All benchmarks, with the exception of Genome, are executed with high and low data contention configurations, as shown in Table 1. Lee's routing algorithm uses early release [3] for its low data contention configuration, which releases unnecessary data from a transaction's read set to reduce false conflicts. This requires application-specific knowledge to determine which data is unnecessary, and manual annotation of the code. The input parameters for the benchmarks are those recommended by their respective providers.

4 Performance Evaluation

In control theory, a controller's performance is primarily measured as its effectiveness to reduce variance in its *output*. This section evaluates the effectiveness of the five CCAs at reducing variance in execution time, resource usage, *wasted work* and *aborts per commit*. Hereafter, *static execution* refers to execution with a fixed number of threads, and *dynamic execution* refers to execution under any CCA.

4.1 Execution Time

For each benchmark, the CCAs should: a) improve execution time over static execution using an initial number of threads that under-exploits the exploitable parallelism, and b) reduce variance in execution time with different numbers of initial threads. Both points are validated in Figure 1. Genome, Lee-TM-ter, and VacH show clear examples of the CCAs improving execution time when the initial number of threads under-exploits the available exploitable parallelism, and reducing variance in execution time irrespective of the initial number of threads. There is little difference in execution time between the CCAs, but only PoCC consistently performs well, whereas SA, EI, EA, and EC all show poor execution times in some benchmark configurations.

Figure 2 presents the execution time standard deviation for each benchmark to compare the effectiveness of the CCAs at reducing execution time variance. The results show PoCC is the best on average, reducing standard deviation by 31% over the next best, EC. Furthermore, averaging speedup of each CCA over static execution for each benchmark configuration, PoCC is second-best with an average speedup of 1.26, and EC is best with a slightly improved speedup of 1.27. Averaging speedup of each CCA over best-case static execution for each benchmark, PoCC is joint-best with EC with an average slowdown of 5%, while EI, EA, and SA suffer an average slowdown of 6%, 7%, and 10%, respectively.

4.2 Resource Usage

Resource usage is calculated by summing, for all samples, the sample duration multiplied by the number of threads executing during the sample. For each

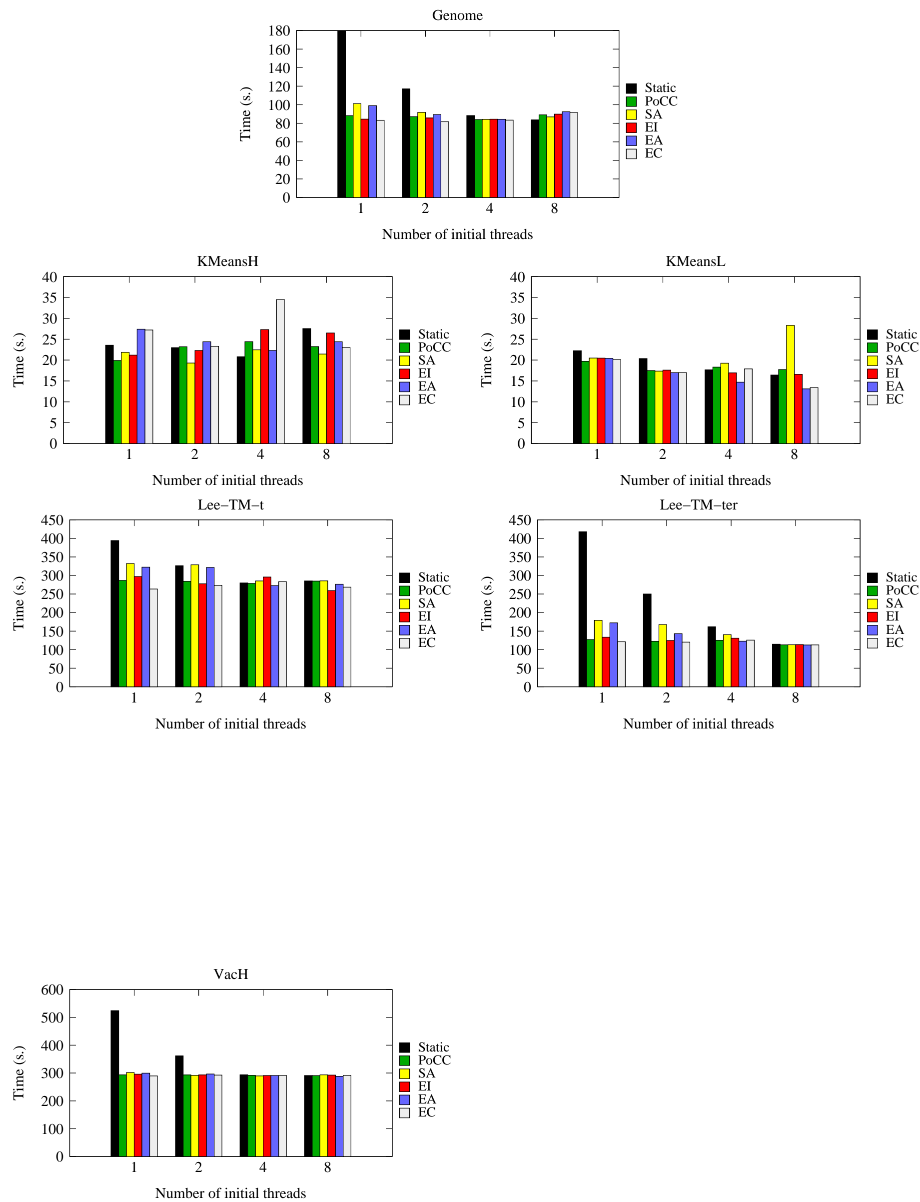

Fig. 1. Execution times for complex benchmarks. StepChange benchmark data is omitted as it executes for a fixed 300 seconds. Less is better.

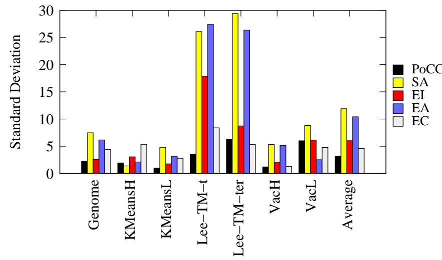

Fig. 2. Execution time std deviation over all initial threads. Less is better.

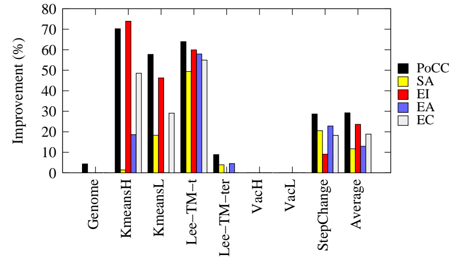

Fig. 3. Resource efficiency vs. static execution at 8 initial threads. More is better.

benchmark, the CCAs should improve resource usage over static execution using an initial number of threads that over-exploits the exploitable parallelism.

We choose to compare resource usage at 8 initial threads for two reasons: 1) applications that scale past 8 threads should show little resource usage improvement, and 2) applications that do not scale past 8 threads should get maximum resource usage saving at 8 threads with dynamic execution, and thus allow direct comparison between the CCAs. Figure 3 presents the resource usage improvement for each CCA, and shows PoCC is the best on average, improving resource savings by 24% over the next best, EC.

4.3 Transaction Execution Metrics

Two transaction execution metrics are presented: *wasted work* and *aborts per commit* (APC). Wasted work is the proportion of execution time spent in executing transactions that eventually aborted, and APC is the ratio of aborted transactions to committed transactions. Both metrics are a measure of wasted execution, and are thus of interest since concurrency control attempts to reduce variance in TCR, which should result in reduced variance in these metrics.

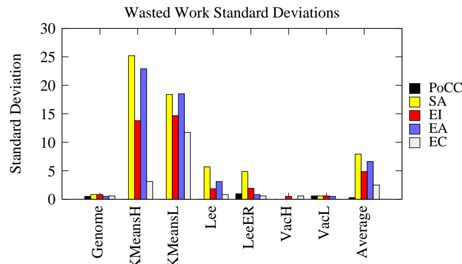

Fig. 4. Wasted work standard deviations for the benchmarks. Less is better.

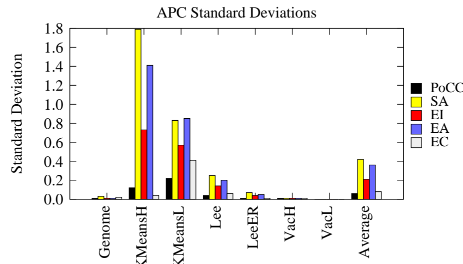

Fig. 5. APC standard deviations for the benchmarks. Less is better.

Figure 4 presents wasted work standard deviations. PoCC significantly reduces variability in wasted work: on average its standard deviation is 88% lower than the next best CCA, which is EC. Figure 5 presents APC, and again PoCC reduces variability: on average its APC standard devation is 26% lower than the next best CCA, which is EC. Furthermore, PoCC reduces average wasted work by 16% over the next best CCA, which is EC, and reduces average APC by 11% over the next best CCA, which is also EC.

4.4 Controller Responsiveness

Controller response is usually measured by three metrics: 1) *response rate*: how fast the number of threads rises when TCR changes, 2) *settle rate*: how quickly the number of threads stops oscillating following the response, and 3) *overshoot*: maximum number of threads above the settled value number of threads.

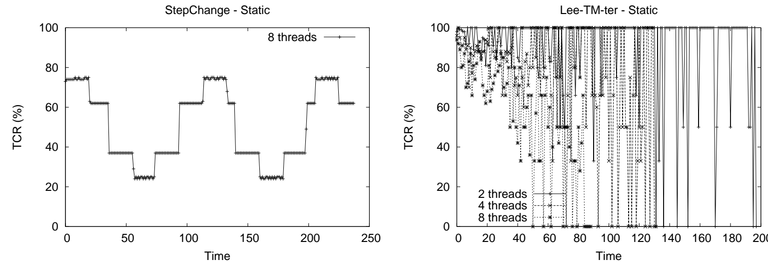

Fig. 6. TCR profiles of StepChange and Lee-TM-ter.

The responsiveness analysis is restricted to StepChange and Lee-TM-ter. Both exhibit TCR profiles that stress the CCAs as shown sampled at 1 second intervals in Figure 6. StepChange changes TCR by large amounts at fixed intervals, and Lee-TM-ter has a wildly oscillating TCR due to a fast sample rate.

Figure 7 presents the CCAs' response graphs. PoCC shows good response to both benchmarks: it responds to StepChange quickly due to the 1 second sample rate, but is also robust to noise in Lee-TM-ter due to the *minTransactions* filter. It has no overshoot or settle rate.

The other four CCAs have a slow sample rate, which gives them no overshoot or settle rate, makes them robust to noise in Lee-TM-ter, but respond poorly in StepChange. This is the trade-off for these four CCAs; responsiveness vs. robustness, as mentioned in Section 2.

5 Conclusion

This work has presented a new concurrency control algorithm, called PoCC, and evaluated it against a synthetic benchmark and several complex benchmarks. An

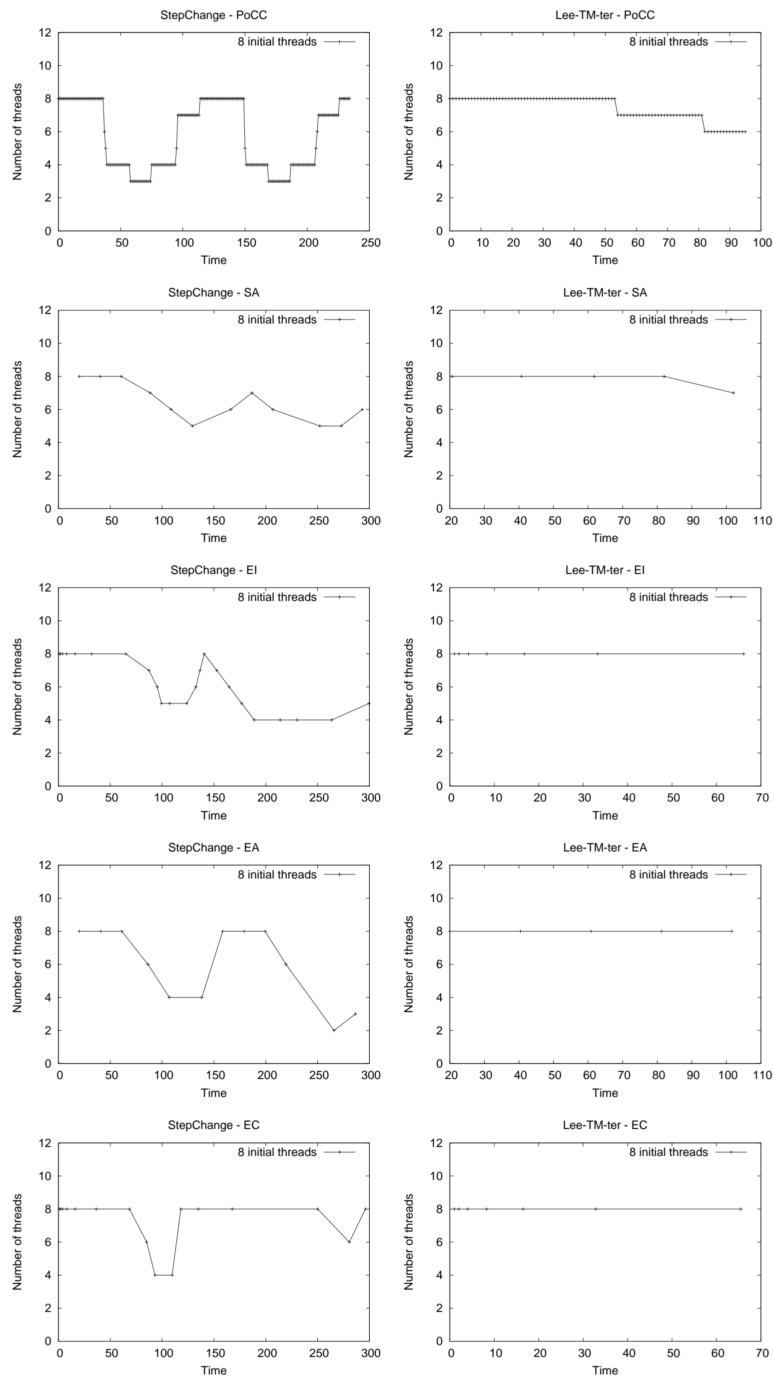

Fig. 7. Number of threads dynamically changing in response to changes in TCR, using all CCAs.

extensive evaluation with several complex benchmarks showed PoCC maintains average execution time similar to the best CCA, has the least performance deficit vs. best-case fixed-thread execution, and improves over the other four CCAs by at least 24% average resource usage, 16% average wasted work, and 11% average APC. PoCC improves over the other four CCAs standard deviation by at least 31% in execution time, 24% in resource usage, 88% in wasted work, and 26% in APC. Thus PoCC matches or improves in all benchmark performance metrics analyzed, and improves controller performance by significantly reducing variability in the benchmark performance metrics.

Finally, an analysis of all the CCAs' response characteristics shows PoCC to be more responsive to, and more robust to noise in, changes in TCR. This is due to the new features in PoCC allowing fine-grain response to changes in TCR, and allowing the sample period to be application-independent.

References

1. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. ACM Press, May 1993.
2. Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, August 1995.
3. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101. ACM Press, July 2003.
4. Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Adaptive concurrency control for transactional memory. In *MULTIPROG '08: First Workshop on Programmability Issues for Multi-Core Computers*, January 2008.
5. Karl Astrom and Tore Hagglund. *PID Controllers: Theory, Design, and Tuning.* Instrument Society of America, 1995.
6. Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 253–262. ACM Press, October 2006.
7. Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proceedings of the 24th Annual Symposium on Principles of Distributed Computing*, pages 258–264. ACM Press, July 2005.
8. William Scherer III and Michael Scott. Contention management in dynamic software transactional memory. In *CSJP '04: Workshop on Concurrency and Synchronization in Java Programs*, July 2004.
9. William Scherer III and Michael Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the 24th Annual Symposium on Principles of Distributed Computing*, pages 240–248. ACM Press, July 2005.

10. Ian Watson, Chris Kirkham, and Mikel Luján. A study of a transactional parallel routing algorithm. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 388–400. IEEE Computer Society Press, September 2007.
11. Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 69–80. ACM Press, June 2007.