Adaptive Concurrency Control for Transactional Memory

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

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

Abstract. Transactional applications may exhibit fluctuating amounts of contention during execution. Excessive numbers of threads executing transactions can produce phases with a high transaction abort ratio. while few threads executing transactions will under-perform in phases with low contention. This paper presents the first application of *adaptive concurrency control* to TM in order to dynamically adjust the number of threads executing transactions concurrently. Four adaptive schemes are implemented in DSTM2, a software TM implementation, and evaluated against a TM application with complex and realistic behavior. Adaptive concurrency control complements existing contention management policies that capture which transaction should be aborted when two transactions conflict.

1 Introduction

The future of processor technology has been confirmed as multicore [1]. Mainstream processor manufacturers have all changed their product line-up to multicore. Multicore processors set a new precedent for software developers: software will need to be multithreaded to take advantage of future processor technology [2] . Furthermore, given that the number of cores is only likely to increase, the parallelism in the software should be abundant to ensure it improves performance on successive generations of multicore processors.

Transactional memory (TM) [3] is a parallel programming abstraction that promises to simplify parallel programming by offering implicit synchronization. Programmers using TM label as *transactions* those portions of code that access shared data, and the underlying TM implementation maintains atomicity, consistency, and isolation. The TM implementation monitors the execution of transactions and *commits* those that do not have access conflicts. For any two transactions that have access conflicts, the TM implementation will *abort* one, and let the other continue executing. Selecting the transaction to abort is determined by a *contention management policy* [4–6].

Performance of TM implementations has been the subject of intense investigation in recent years. This paper studies techniques that complement contention management policies to improve performance and resource utilization that can be easily applied to TM implementations: adaptive concurrency control.

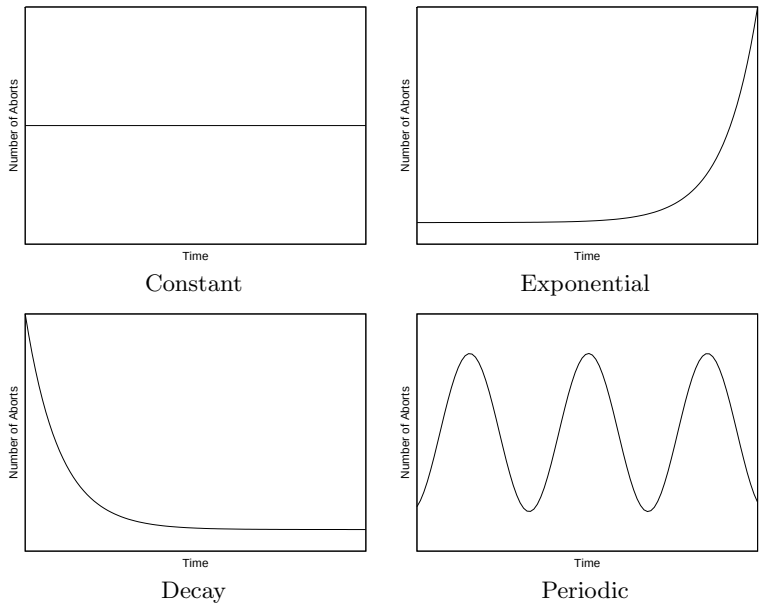


Fig. 1. Example patterns of contention fluctuation over the execution time of an application.

Figure 1 shows example patterns of fluctuating contention (number of aborts) that transactional applications may exhibit during execution. Running applications with such dynamic contention using a fixed number of threads can hurt performance and be resource inefficient. Excessive numbers of threads during phases with high contention hurt performance by increasing the number of conflicts, which in turn wastes resources through aborted transactions. Similarly, a limited number of threads will under-perform in phases with low contention.

Existing TM research has not investigated dynamic contention levels for performance or resource usage improvements. *Adaptive concurrency control*, which dynamically adjusts the number of threads executing concurrently, aims to take advantage of fluctuating contention to improve performance and resource utilization.

This paper presents the first application of adaptive concurrency control to TM. Four schemes are implemented in DSTM2 [7], a software TM implementation, and evaluated against a recently published TM application [8] with complex and realistic behavior that exhibits fluctuating contention during execution. Using 8 threads, an average performance improvement of 38% and resource usage improvement of 53% is achieved.

The rest of this paper is organized as follows. Section 2 introduces the four adaptive concurrency control schemes. Section 3 introduces the experimental platform and the application used to evaluate the schemes including a brief description of the considered contention policies. Sections 4 and 5 present the

results of using adaptive concurrency control on the application in terms of performance and resource utilization, respectively. Section 6 discusses related work, and Section 7 concludes the paper.

2 Adaptive Concurrency Control

Feedback-based control has a long history of application in a diverse range of fields, inside and outside computing, to maintain some variable within a bounded range. This paper targets *Transaction Commit Ratio* (TCR), the percentage of committed transactions in the total number of transactions executed, as that variable for transactional memory. Using TCR to control the number of threads is motivated by the fact that TCR falls during phases with high contention, which indicates the number of threads can be reduced, and vice versa.

Adaptive concurrency control removes the need for a user to specify the number of threads with which an application should be executed. This number is typically discovered through trial and error, is specific to a certain software/hardware combination, may require reassessment every time the application is changed, and may still be suboptimal if the amount of contention in the application fluctuates during execution. Adaptive concurrency control simply adjusts the number of threads to what is best suited for the application based on its TCR.

Finally, long transactions are a known difficulty for TM as they can be constantly aborted by shorter transactions, leading to starvation. Adaptive concurrency control based on TCR can address this problem, when there are enough long transactions executing concurrently to significantly reduce the TCR. This will cause the number of threads to keep decreasing (possibly down to one thread) until the TCR rises again, i.e. when the long transactions are committing.

Adaptive concurrency control has two parameters: the *target TCR range*, and the *sample interval* over which the TCR is sampled in order to make a concurrency control decision. Below, four adaptive concurrency control schemes are described which vary in the strength of their response to the change in TCR. Whilst the schemes are prototypes, they are loosely similar to multivariable PID controllers [9] used in control theory.

2.1 SimpleAdjust

SimpleAdjust is the simplest scheme that increments the number of executing threads by one if the TCR is above the upper TCR threshold. Similarly, the number is reduced by one if the TCR is below the lower threshold. When the TCR is within the target range, no change is made.

2.2 ExponentialInterval

ExponentialInterval extends SimpleAdjust with the aim of improving response time to TCR changes. If a change to the number of threads is made then the sample interval is halved, i.e. the next change, if necessary, will be made sooner.

Conversely, the sample interval is doubled if no change has been made to the number of threads. As before, the number of executing threads is only increased or decreased by one.

2.3 ExponentialAdjust

ExponentialAdjust is another extension to SimpleAdjust that aims to improve the response to a change in TCR. ExponentialAdjust keeps the sample interval fixed, and calculates the adjustment to the number of executing threads based on the difference of the sample TCR and the target TCR range. The further the sampled TCR is from the target TCR range, the greater the adjustment. The formula initially chooses to add one thread, and then doubles this value for every 10% the TCR is outside the target TCR range. For example, using a target TCR range of 30–60% and a sampled TCR of 80%, ExponentialAdjust would add four threads.

2.4 ExponentialCombined

ExponentialInterval and ExponentialAdjust are two orthogonal approaches to improving the responsiveness to the change in TCR. ExponentialCombined combines the sample interval adjustment of ExponentialInterval, and the variable thread adjustment of ExponentialAdjust, resulting in the most responsive adaptive concurrency control scheme.

3 Experimental Platform

This section begins by describing the Software TM (STM) implementation used, and the modifications that enable the adaptive schemes to work. A brief overview of the application used is given, followed by implementation details of the adaptive schemes. Finally, the hardware platform and the experimental configurations used to gather results are presented.

3.1 STM implementation

The STM used for experimental analysis of the adaptive schemes is the Java-based DSTM2 [7]. Although several STM implementations have been published [10, 11], DSTM2 was chosen for its ease of use, popularity, and diverse set of contention managers — analyzing the adaptive schemes against several contention managers allows greater scrutiny. The contention managers are Backoff, Aggressive, Eruption, Greedy, Karma, Kindergarten, Priority, and Polka. They are described briefly in Section 3.2, for further details refer to [4–6].

A lightweight data sampling mechanism was implemented for DSTM2 to gather data needed by the adaptive schemes to make their decisions. Threads in DSTM2 collect simple statistics locally, and the data sampling mechanism collects this data from the executing threads into a central location. Over several test runs there was no noticeable loss in performance as a result of performing the data sampling.

3.2 Contention Managers

In DSTM2, a contention manager is invoked by a transaction when it finds itself in conflict with another transaction or set of transactions. The contention manager decides which transaction(s) should be aborted based on its policy. There are eight contention managers (policies) implemented in DSTM2. Brief descriptions of each contention manager follow.
Backoff gives the enemy transaction exponentially increasing amounts of time to commit, for a fixed number of iterations, before aborting it.
Aggressive always aborts a conflicting enemy transaction.
Karma gives dynamic priorities to transactions based on the number of objects they have opened for reading, and aborts enemy transactions with lower priorities.
Eruption, like Karma, assigns dynamic priorities to transactions based on the number of transactional objects they have opened for reading. Conflicting transactions with lower priorities add their priority to their opponent to increase the opponent's priority, and allow the opponent to abort its enemies, and 'erupt' through to commit stage.
Greedy aborts the younger of the conflicting transactions, unless the older one is suspended or waiting, in which case the older one is aborted.
Kindergarten works by making transactions abort themselves when they meet a conflicting transaction for the first time, but then aborting the enemy transaction if it is encountered in a conflict a second time.
Priority is a static priority-based manager, where the priority of a transaction is its start time, that aborts lower priority transactions during conflicts.
Polka combines Karma and Backoff by giving the enemy transaction exponentially increasing amounts of time to commit, for a number of iterations equal to the difference in the transactions' priorities, before aborting the enemy transaction.

3.3 Application: Transactional Routing

This application is a recently published complex TM application [8] based on Lee's routing algorithm [12], one of the first complex applications designed to stress TM systems. Routing is used to automatically map printed circuit boards (PCBs) in electronic design. Routing is performed in two phases: an expansion phase that searches outwards from the source point to the destination point on the PCB grid, and a backtrack phase that marks the route onto the PCB by going backwards from the destination to the source. Routing is attempted in parallel, where the laying of each route is a transaction. This provides a mix of long and short transactions.

The routes are read from a file that contains source and destination points for each route as pairs of x and y coordinates, and then sorted in ascending length order into a work queue used by the transactional threads. The circuit routed by the adaptive schemes is shown in Figure 2. This is a realistic circuit that contains 1506 routes and has been used in routing algorithm research.

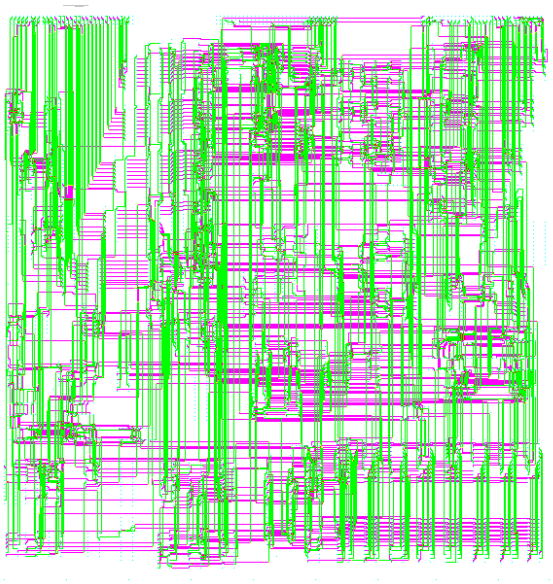


Fig. 2. Circuit routed by the TM application.

3.4 Adaptive System Configuration

DSTM2 maintains a thread pool, and when an adaptive technique decides to decrease the number of existing threads it flags threads to pause rather than terminating them. The threads poll the flag on each commit or abort of a transaction and, if set, exit their run loop safely. The adaptive schemes are never made aware of the number of physical processors available.

As mentioned before the adaptive schemes need two parameters: sample interval and target TCR range. Through experimentation these were set to a sample interval of 20 seconds, lower TCR threshold of 30% and upper threshold 60%.

ExponentialInterval and ExponentialCombined dynamically change the sample interval, but this is bounded to a minimum of 4 seconds to prevent oversensitivity, and maximum of 60 seconds to prevent unresponsiveness.

3.5 Hardware Platform & Benchmark Configurations

The experimental platform used is an 8-way machine with four dual-core 2.4GHz Opteron processors, 16GB RAM, running openSuSE 10.1, and all experiments were run on 64-bit Sun Java6 build 1.6.0-b105 with the flags -Xms1024m -Xmx4096m.

The benefit of the four adaptive schemes described earlier is evaluated against non-adaptive — hereafter referred to as NonAdaptive — runs, where each run consists of: adaptive scheme, contention manager, and initial number of threads: 1, 2, 4, or 8. Each run is repeated three times, and the best time is used.

4 Performance Results

Table 1 shows the speedup of the adaptive schemes. The first observation is that, on average, the adaptive schemes offer improvements in the range 11–18%. At 8 threads only one contention manager, Priority, suffers a performance loss with all adaptive schemes, while the average performance improvement is at least 34%. Furthermore, half of the contention managers experience significant performance improvements; over 30% using Eruption, Greedy, and Karma, and astounding 2- to 3-fold speedups using Backoff. This suggests that at this level of parallelism, there are some phases of execution where the contention is high enough that the adaptive schemes have a visible effect on performance. With fewer threads, although such phases of high contention may have occurred, they were not significant enough to cause a performance degradation using NonAdaptive, and in turn show a performance improvement using adaptive schemes.

The large performance benefit of adaptive schemes with the Backoff contention manager at 8 threads is also due to another problem that the adaptive schemes help to mitigate: long transactions. As mentioned previously the application sorts the routes in ascending length order, and as a result all the longest routes get executed concurrently near the end of the application's run. Long routes are far more likely to conflict than short routes, and Backoff's policy is to give the opposing transaction some time to complete before aborting it. This allows a situation to occur where two routes are long enough that their execution time leads them to aborting one another. The adaptive schemes responded to the fall in TCR at that stage, and resulted in much better performance for Backoff.

Aggressive, Kindergarten and Priority, at 8 threads, have the best raw performance results for NonAdaptive, showing that these contention managers are suffering the least from contention issues. Adaptive schemes improve the performance of two of these (Aggressive and Kindergarten) by 1–10%, showing that the adaptive schemes are not only useful when contention is significantly high. Note that Aggressive and Kindergarten are improved by 6% when combined with SimpleAdjust.

Another observation is that the adaptive schemes are interchangeable in terms of average performance, with none offering significant advantages (14–18%). The bottom row in Table 1 shows the speedup values of each technique averaged over all its runs, and confirms that there is very little difference in performance between the schemes. This is likely due to the application not exhibiting large and frequent fluctuations in transactional contention, and thus not allowing the schemes with faster responses to offer better performance.

The graphs also show that the performance of the adaptive schemes varies depending on the number of threads with which the application is initialized, though the schemes are still more stable than the NonAdaptive runs. Thus, the adaptive schemes still require further tuning before the need to specify the number of threads can be completely removed.

	SimpleAdjust				ExponentialInterval				ExponentialAdjust				ExponentialCombined				**CM**
Contention Manager	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8	**Average**
Aggressive	0.94	1.24	0.94	1.06	0.92	1.13	1.00	1.07	1.01	1.25	1.07	1.10	1.08	1.18	1.03	1.04	**1.07**
Backoff	0.82	0.74	1.63	2.47	0.84	0.87	1.39	2.73	0.76	0.90	1.41	3.00	0.89	0.91	1.41	2.47	**1.45**
Eruption	0.72	1.14	1.12	1.42	0.82	1.13	1.03	1.39	0.81	1.21	0.95	1.49	0.83	1.21	0.93	1.52	**1.11**
Greedy	1.20	1.08	1.00	1.34	0.99	0.98	1.00	1.26	1.14	1.04	1.00	1.36	1.08	0.99	0.94	1.33	**1.11**
Karma	1.12	1.04	1.05	1.31	1.02	1.21	1.05	1.30	1.18	1.13	1.04	1.41	1.05	1.13	1.03	1.41	**1.16**
Kindergarten	1.12	1.18	0.99	1.06	1.13	1.07	0.91	1.02	1.30	1.22	0.99	1.05	1.35	1.14	0.99	1.01	**1.10**
Polka	0.96	1.23	0.97	1.08	1.01	1.03	0.94	1.09	1.07	1.09	1.08	1.24	1.04	1.02	0.92	1.14	**1.06**
Priority	1.32	1.09	1.05	0.98	1.13	0.95	1.04	0.98	1.21	1.08	1.04	1.00	1.23	1.05	1.04	0.98	**1.07**
Thread Average	**1.02**	**1.09**	**1.09**	**1.34**	**0.98**	**1.05**	**1.05**	**1.36**	**1.06**	**1.11**	**1.07**	**1.46**	**1.07**	**1.08**	**1.04**	**1.36**	
Scheme Average	**1.14**				**1.11**				**1.18**				**1.14**				

Table 1. Speedup over NonAdaptive for each adaptive scheme and each contention manager.

5 Resource Utilization Results

The previous section presented the performance results of the adaptive schemes, and showed that there was little difference in performance among them, and thus, for brevity, this section only discusses the resource utilization of one of the schemes, SimpleAdjust. The results also show that benefits were appearing significantly at 8 threads. Thus, again for brevity, only the resource usage data of the 8 thread runs is presented. Figure 3 shows the resource utilization of SimpleAdjust compared to NonAdaptive for each contention manager. The data shown is the number of threads executing concurrently at intervals during the execution of the application.

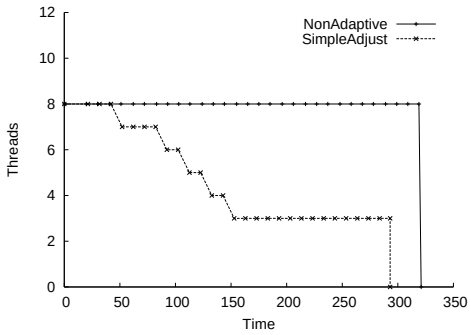
All the graphs show an exponential decay, which is consistent with the application's operation: it lays routes in ascending length order so as the execution progresses, the amount of contention is expected to increase, reducing the TCR. The results show a clear improvement in resource usage, with an average reduction of 53% (see Table 2) and reductions in the range 41–82%. As mentioned in the previous section, the performance has improved in all cases except for the Priority contention manager.

Contention Manager	Improvement (%)	Contention Manager	Improvement (%)
Aggressive	46	Backoff	82
Eruption	59	Greedy	57
Karma	53	Kindergarten	44
Polka	41	Priority	41

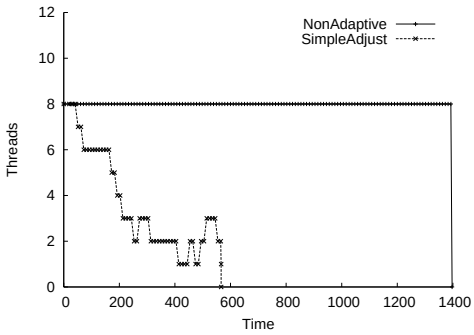
Table 2. Resource utilization improvement using SimpleAdjust compared to NonAdaptive using 8 threads. Average improvement: 53%

6 Related Work

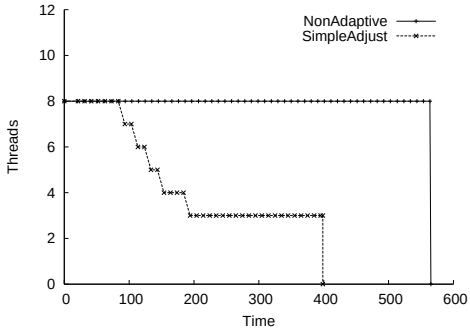
This is the first paper considering adaptive concurrency control for TM, but Marathe *et al.* [13] have investigated adapting other TM components. They designed and evaluated a STM implementation, called ASTM, that adapts between eager and lazy data acquisition, and adapts between direct and indirect object referencing. Their results showed that ASTM yields throughput that is comparable with the best STM implementations across a range of benchmarks, whereas previously certain STMs would be markedly better at executing certain benchmarks. Their techniques are orthogonal to ours, and could be easily combined to produce a more sophisticated adaptive STM implementation. Both their adaptive techniques and our adaptive techniques are general-purpose and not application or implementation specific.



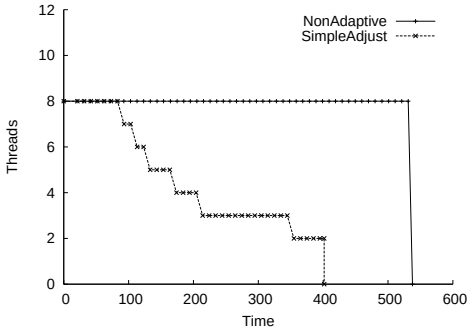
Resource utilization with Aggressive.



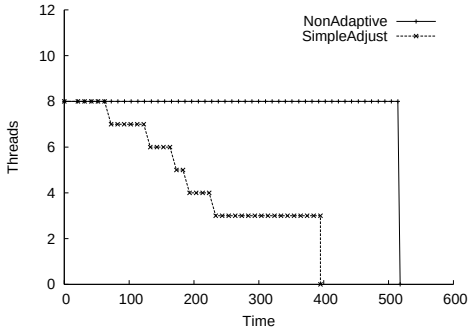
Resource utilization with Backoff.



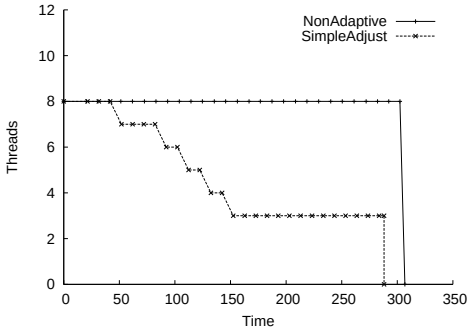
Resource utilization with Eruption.



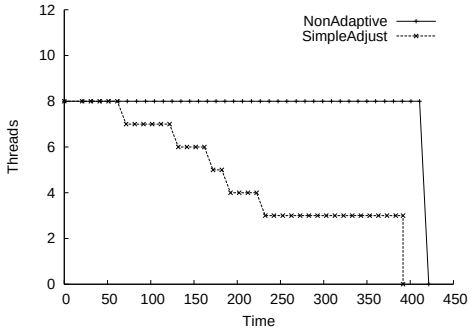
Resource utilization with Greedy.



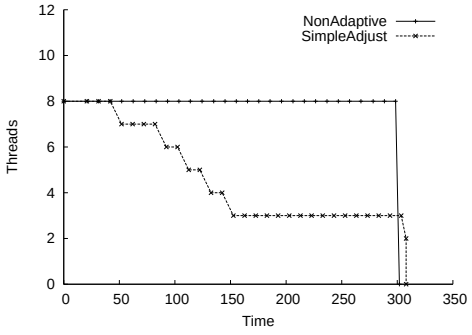
Resource utilization with Karma.



Resource utilization with Kindergarten.



Resource utilization with Polka.



Resource utilization with Priority.

Fig. 3. Resource usage during execution for SimpleAdjust, with each contention manager.

7 Conclusions

This paper has presented the first application of adaptive concurrency control in TM with the aim of improving resource utilization and performance by reducing contention. Four adaptive schemes were implemented for DSTM2 [7] that adjust the number of threads executing concurrently in response to a change in the transaction commit ratio (TCR) with various response strengths. They are compatible with, and complement, existing contention management policies.

Evaluation against a complex and realistic TM application showed significant resource usage and modest performance improvements. At 8 threads, in the average case adaptive concurrency control led to a performance improvement of 38% and resource usage improvement of 53%, and in the worst case performance dropped 2% and resource usage improved 41%.

The adaptive schemes can be easily applied to other STM implementations to take advantage of their benefits as they do not take advantage of any low-level software- or hardware-specific optimizations.

References

1. Kunle Olukotun and Lance Hammond. The future of microprocessors. *ACM Queue*, 3(7):26–29, 2005.
2. Richard McDougall. Extreme software scaling. *ACM Queue*, 3(7):36–46, 2005.
3. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th Annual iIternational Symposium on Computer Architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.
4. William Scherer III and Michael Scott. Contention management in dynamic software transactional memory. In *CSJP '04: Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, Jul 2004.
5. 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, New York, NY, USA, 2005. ACM Press.
6. 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, New York, NY, USA, 2005. ACM Press.
7. 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, New York, NY, USA, 2006. ACM Press.
8. Ian Watson, Chris Kirkham, and Mikel Lujan. 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, September 2007.
9. Karl Astrom and Tore Hagglund. *PID Controllers: Theory, Design, and Tuning.* Instrument Society of America, Research Triangle Park, NC, USA, 1995.

10. Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, New York, NY, USA, 2006. ACM Press.
11. David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*, pages 194–208, 2006.
12. F. Rubin. The Lee path connection algorithm. *IEEE Transactions on Computers*, C-23(9):907–914, 1974.
13. Virendra Marathe, William Scherer III, and Michael Scott. Adaptive software transactional memory. In *DISC '05: Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sep 2005.