Robust Adaptation to Available Parallelism in Transactional Memory Applications

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

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

Abstract. Applications using transactional memory may exhibit fluctuating (dynamic) available parallelism, i.e. the maximum number of transactions that can be committed concurrently may change over time. Executing large numbers of transactions concurrently in phases with low available parallelism will waste processor resources in aborted transactions, while executing few transactions concurrently in phases with high available parallelism will degrade execution time by not fully exploiting the available parallelism. Three questions come to mind: (1) Are there such transactional applications? (2) How can such behaviour be exploited? and (3) How can available parallelism be measured or calculated efficiently? The contributions of this paper constitute the answers to these questions.

This paper presents a system, called transactional concurrency tuning, that adapts the number of transactions executing concurrently in response to dynamic available parallelism, in order to improve processor resource usage and execution time performance. Four algorithms, called controller models, that vary in response strength were presented in previous work and shown to maintain execution time similar to the best case non-tuned execution time, but improve resource usage significantly in benchmarks that exhibit dynamic available parallelism.

This paper presents an analysis of the four controller models' response characteristics to changes in dynamic available parallelism, and identifies weaknesses that reduce their general applicability. These limitations lead to the design of a fifth controller model, called P-only transactional concurrency tuning (PoCC). Evaluation of PoCC shows it improves upon performance and response characteristics of the first four controller models, making it a robust controller model suitable for general use.

1 Introduction

The future of processor architectures has been confirmed as multi-core [1–3], and mainstream processor manufacturers have all changed their product line-up. Multi-core processors set a new precedent for software developers: software will need to be multi-threaded to take advantage of future processor technology [4]. Furthermore, given that the number of cores is only likely to increase, the

parallelism in the software should be abundant to ensure it continues to improve performance on successive generations of multi-core processors.

Transactional Memory (TM) [5–7] is a programming abstraction that promises to simplify parallel programming by offering implicit synchronisation. Programmers using TM label as *transactions* those portions of code that access shared data, and the underlying TM ensures safe access. The TM implementation monitors the execution of transactions, and for any two transactions that have access conflicts the TM implementation will *abort* one, and let the other continue executing. A transaction *commits* if it does not have access conflicts, thus making its updates to shared data available to the rest of the application.

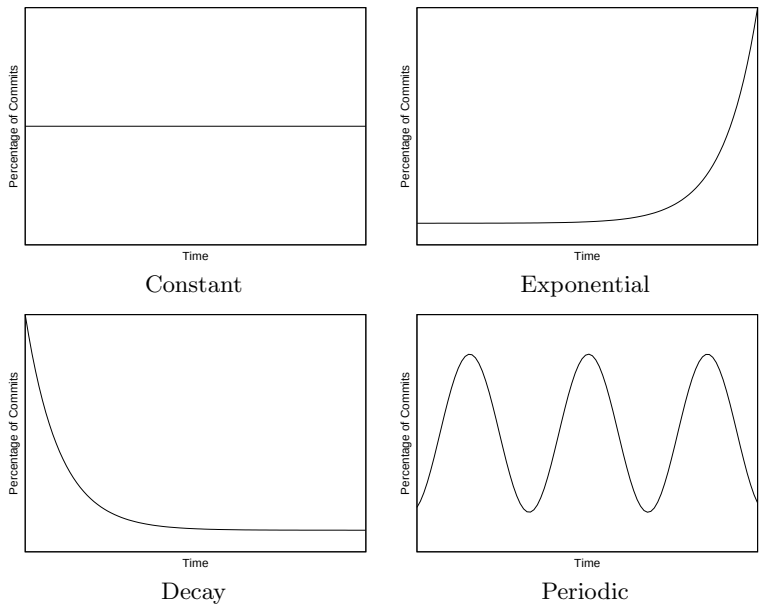

Fig. 1. Example patterns of available parallelism, expressed as a percentage of concurrently attempted transactions that commit.

Figure 1 shows examples of fluctuating available parallelism patterns that transactional applications may exhibit during execution. We define available parallelism as *the maximum number of transactions that can be committed concurrently*, i.e. none aborting. Executing applications that have dynamic available parallelism with a fixed number of concurrent transactions can hurt performance and be resource inefficient. Executing large numbers of transactions concurrently in phases with low available parallelism a) wastes resources in the execution of aborted transactions, b) hurts performance by increasing the number of access conflicts that have to be resolved, and c) hurts performance and wastes resources when aborted transactions need to be rolled back. Similarly, executing too few

transactions concurrently in phases with high available parallelism hurts execution time performance.

This paper studies a technique to take advantage of dynamic available parallelism. *Transactional concurrency tuning* dynamically adjusts the number of transactions executing concurrently with respect to the available parallelism, in order to improve execution time performance and resource usage efficiency. We identify *Transaction Commit Rate* (TCR), which is the percentage of committed transactions out of all executed transactions in a sample period, as a suitable lightweight, application-independent measure of available parallelism.

Four controller models that vary in their response strength to changes in available parallelism are implemented, and evaluated. Following an investigation of their response characteristics, a fifth controller model was implemented that combines the strengths of the four models and improves their response characteristics. Evaluations are performed using a synthetic benchmark, Lee-TM [8, 9], and STAMP [10] applications Genome, KMeans, and Vacation, which have become popular [11–16] non-trivial benchmarks in TM research. Evaluation is carried out using DSTM2 [17], a software TM (STM) implementation.

The paper is organised as follows: Section 2 introduces transactional concurrency tuning and the controller models. Section 3 evaluates the controller models. Section 4 discusses the applicability and effectiveness of transactional concurrency tuning. Section 5 concludes the paper.

2 Transactional Concurrency Tuning

Transactional concurrency tuning has its origins in control theory, which is widely used in industrial processes to maintain system parameters at user-defined optima. Defining transactional concurrency tuning for TM using control theory terminology, the control objective is to maintain the *process variable* TCR at a *setPoint* desirable value, in spite of *unmeasured disturbance* from fluctuating available parallelism. TCR and *setPoint* are percentage values in the range 0–100%. The *setPoint* determines how conservative a controller model is towards resource usage efficiency. A high *setPoint*, e.g. 90%, causes a controller model to be quick to reduce threads when TCR decreases, but slow to adapt to a sudden large increase in TCR, and vice versa. Transactional concurrency tuning also has a parameter called *samplePeriod* over which the TCR is sampled in order to make a transactional concurrency tuning decision. The controller models set this parameter in different ways.

The *controller model output* is to modify the number of threads executing transactions in response to changes in TCR. In order to do this, a thread pool framework is implemented to execute transactions, and the controller model output changes the number of threads active in the thread pool. Each worker thread has its own work queue, as the traditional single work queue architecture can quickly become a bottleneck. The worker threads also implement work stealing [18] to reduce load imbalance. Application threads submit jobs to the thread pool, and submission can be either synchronous, i.e. the application thread waits

until the transaction commits, or asynchronous, i.e. the application thread submits the job, but does not wait for the transaction to commit. Using only synchronous submission emulates the existing TM programming model, but asynchronous submission may improve exploitation of high available parallelism; if the number of worker threads is increased such that it is greater than the number of application threads, synchronous submission will not deliver enough jobs for all worker threads. Figure 2 illustrates how the transaction concurrency tuning system permits modular controller models, i.e. the policy for determining the *controller model output*.

1. if `currentTime` − `lastSampleTime` < *samplePeriod*, goto Step 1;
2. `TCR` ← `numCommits` / `numTransactions` × 100;
3. Δ`threads` ← *controller model output*
4. `newThreads` ← `numCurrentThreads` + Δ`threads`;
5. Adjust `newThreads` such that *minThreads* ≤ `newThreads` ≤ *maxThreads*;
6. `numCurrentThreads` ← `newThreads`;
7. Set `lastSampleTime` ← `currentTime`, go to Step 1;

Fig. 2. Transactional concurrency tuning pseudocode, with modular controller model.

2.1 Four Controller Models

This section introduces four controller models from the authors' previous work [19] that vary in their response strength to the difference in the measured value of the process variable and the *setPoint*. Preliminary experimental analysis found the controller models described below to have unstable controller model output using a single value for the *setPoint* (e.g. 70%) so a *setPointRange* (e.g. 50–80%) is selected.

SimpleAdjust is the simplest controller model, and increments the number of worker threads by one if the sampled TCR is above the upper *setPointRange* value, or vice versa. When the TCR is within *setPointRange*, no change is made.

ExponentialInterval extends SimpleAdjust aiming to improve response time to TCR changes. If a change to the number of worker threads is made then *samplePeriod* is halved, i.e. the next change, if necessary, will be made sooner. Conversely, *samplePeriod* is doubled if the number of worker threads is left unchanged. As before, the number of worker threads is only increased or decreased by one. A *samplePeriodRange* that restricts the *samplePeriod* must be defined.

ExponentialAdjust also extends SimpleAdjust aiming to improve response time to TCR changes. It calculates the adjustment to the number of worker threads based on the difference in sampled TCR and the *setPointRange*. The further the

sampled TCR from the *setPointRange*, the greater the adjustment. The formula initially chooses to add or subtract one worker thread, and then doubles this value for every 10% the TCR is outside the *setPointRange*. For example, using a *setPointRange* of 50–60% and a sampled TCR of 82%, ExponentialAdjust calculates a TCR difference of 22%, and thus doubles the number of threads twice ($1 \rightarrow 2 \rightarrow 4$) to add four worker threads.

ExponentialCombined is a combination of ExponentialInterval and ExponentialAdjust. ExponentialCombined has the sample interval adjustment of ExponentialInterval, and the variable worker thread adjustment of ExponentialAdjust, resulting in the most responsive controller model.

2.2 P-only Controller Model

This section begins by describing the fifth controller model, called P-only transactional concurrency tuning (PoCC), then goes on to discuss the features introduced to make it more general purpose than the four controller models described previously. Specifically, PoCC adds two enhancements: a proportional gain formula, and minimum transaction count filter. PoCC is based on a P-only controller model [20] and is presented as pseudocode in Figure 3.

1. If `numTransactions` $<$ *minTransactions*, goto Step 1;
2. Δ`TCR` $\leftarrow$ `TCR` $-$ *setPoint*;
3. If (`numCurrentThreads`$= 1$) & (`TCR` $>$ *setPoint*);
   (a) then Δ`threads` $\leftarrow 1$;
   (b) else Δ`threads` $\leftarrow$ Δ`TCR` $\times$ `numCurrentThreads` / 100
   (rounded to the closest integer);

Fig. 3. PoCC controller model pseudocode.

In step 1, a new parameter *minTransactions* is added that acts as a filter against noisy TCR profiles such as in Figure 12. Such noisy samples may occur due to the average transaction's duration being longer than the *samplePeriod*. The four previous controller models, lacking PoCC's filter, absorbed noise by using a large *samplePeriod*, which was a trade-off of responsiveness for robustness. PoCC's filter allows it to be highly responsive, by using a short *samplePeriod*, but still be robust to noisy samples. Thus, in PoCC, *samplePeriod* is determined based on the overhead of executing the control system loop, and does not have to filter noisy samples.

The first four controller models used an absolute gain formula to calculate Δ`threads`, which led to a change in `numCurrentThreads` even if small Δ`TCR` values occurred. Such a response was disproportionate at low worker thread counts, e.g. an increase from 1 thread to 2 threads for a TCR only 1% higher than the `setPoint`. This unstable behaviour was controlled by using a *setPointRange*.

However, over large worker thread counts, a *setPointRange* range results in poor responsiveness as it produces coarse-grain control. In step 3(b) PoCC uses a proportional gain formula (i.e., proportional to the number of current worker threads) that allows, in response to small Δ`TCR`, Δ`threads` to be zero at low worker thread counts, and fine-grain control at large worker thread counts. Thus, PoCC improves responsiveness, because its proportional gain formula allows it to use a *setPoint* rather than a *setPointRange*, and still result in stable behaviour at low worker thread counts.

3 Evaluation of the Controller Models

The evaluation is split into several sections: execution time, resource usage, transaction execution metrics, and finally an investigation of controller model response characteristics. The controller models are abbreviated to SA, EI, EA, EC, and PoCC, respectively.

Hereafter, *static execution* refers to execution with a fixed number of threads, and *dynamic execution* refers to execution under any controller model. All experiments use the thread pool to execute transactions. All benchmarks are executed using 1, 2, 4, and 8 *initial threads*. We use the term initial threads as dynamic execution may change the number of threads (between 1 and 8) at runtime. Experiments are executed five times, and the mean results reported. This paper restricts the number of worker threads in the thread pool to a maximum of 8, which is equal to the number of cores in the hardware platform used in the evaluations, and a minimum of 1. Unless specified, references to changing numbers of threads imply thread pool worker threads, and not application threads.

3.1 Controller Model Parameters

Through preliminary experimentation with LeeH (explained later in Section 3.3) the parameters of the first four controller models were set to: *samplePeriod* of 10 seconds, *setPointRange* of 50–80%, and *samplePeriodRange* of 4–60 seconds. PoCC's parameters are: *setPoint* of 70%, *minTransactions* of 100. Experimental evaluation found execution of the complete control loop took on average 2ms with PoCC, thus *samplePeriod* is set to 1 second for PoCC to make its overhead negligible.

3.2 Hardware & Software Platform

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`.

DSTM2 is used with its default configuration of eager validation, visible readers, and shadow atomic factory. DSTM2 has been modified to maintain a thread pool as described earlier. DSTM2 supports a number of contention managers (CMs). In DSTM2, a CM is invoked by a transaction when it finds

itself in conflict with another transaction. The CM decides which transaction should be aborted based on its policy. The CMs used in this paper are described briefly below, and for further details refer to [21–23].

Aggressive always aborts a conflicting enemy transaction.

Backoff gives the enemy transaction exponentially increasing amounts of time to commit, for a fixed number of iterations, before aborting it.

Karma assigns dynamic priorities to transactions based on the number of objects they have opened for reading, and aborts enemy transactions with lower priorities.

Eruption is similar to Karma, and 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 makes 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.

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.

Priority is a static priority-based manager, where the priority of a transaction is its start time, that immediately aborts lower priority transactions during conflicts.

3.3 Benchmarks

One synthetic and seven real, non-trivial benchmark configurations are used in this paper. The synthetic benchmark, StepChange, oscillates the TCR from 80% to 20% in steps of 20% every 20 seconds, 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.

The non-trivial benchmarks used are Lee's routing algorithm [8], and the STAMP [10] benchmarks Genome, KMeans, and Vacation, from STAMP version 0.9.5, all ported to execute under DSTM2. All benchmarks, with the exception

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
LeeL	Lee-TM low contention	early_release:true, file:mainboard
LeeH	Lee-TM high contention	early_release:false, file:mainboard

Table 1. Benchmark configuration parameters used in the evaluation.

of Genome, are executed with high and low data contention configurations, as shown in Table 1. Lee's routing algorithm uses early release [24] 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. In some other publications, e.g. [8], LeeH is referred to as Lee-TM-t, and LeeL is referred to as Lee-TM-ter. The input parameters for the benchmarks are those recommended by their respective providers. The average benchmark execution times are shown in Table 2, and dynamic available parallelism in Figure 4, both using the Priority contention manager.

The benchmarks have been modified to use the thread pool to execute transactions. Only KMeans partially uses synchronous job submission as part of each thread's code executes two transactions, the second of which needs a return value from the first. The remaining benchmarks use asynchronous job submission. Lee-TM and Vacation create all jobs during benchmark initialisation, which is excluded from the recorded execution time. Genome and KMeans create jobs dynamically, and thus include job creation time in the recorded execution time. Jobs are submitted in a round-robin manner to the multiple work queues.

3.4 Execution Time

Execution time results are presented in two parts. First, LeeH is investigated using all CMs, but only with the first four controller models. The motivation

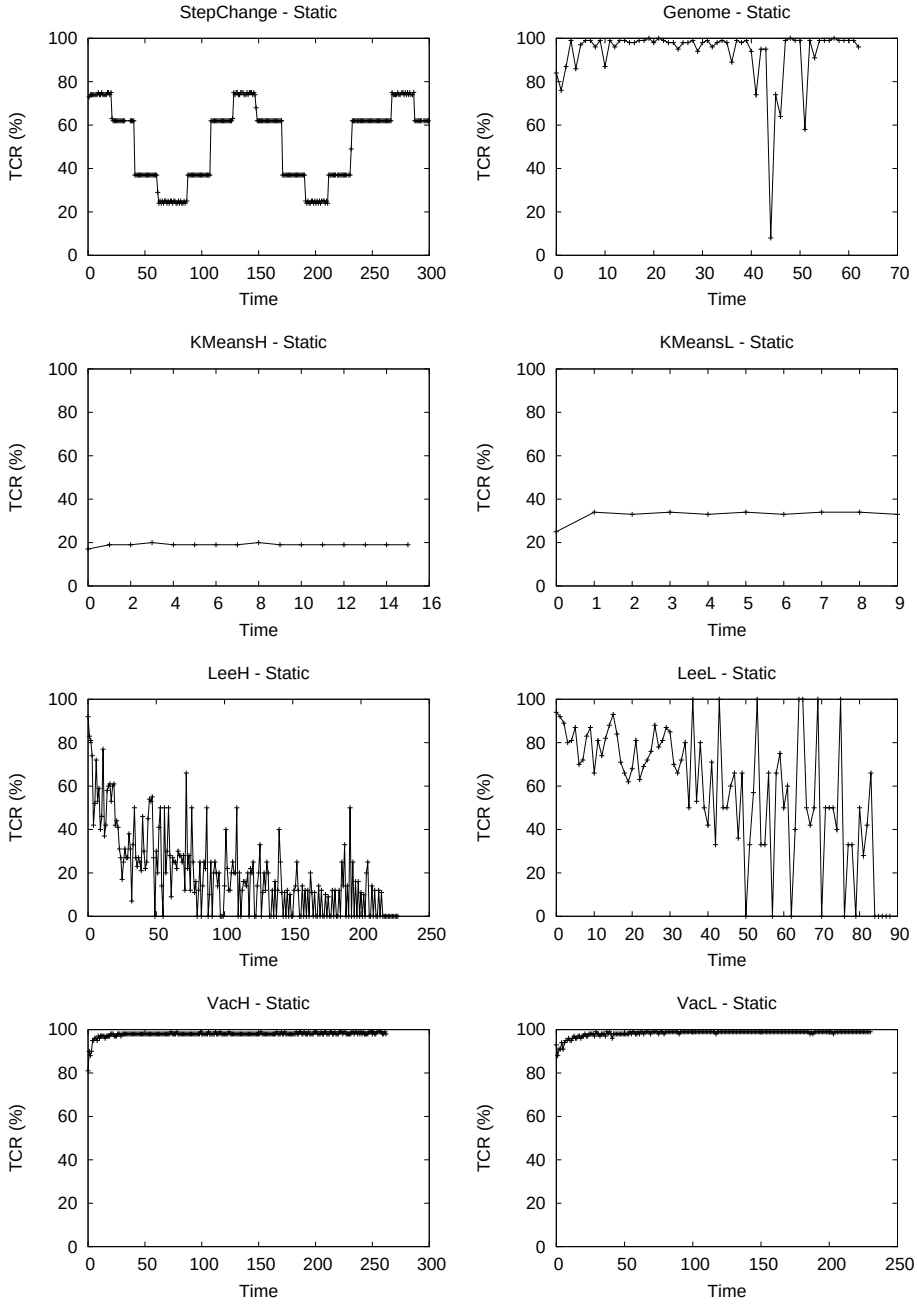

Fig. 4. TCR profiles of the benchmarks used, executing with 8 threads.

	Genome	KMeansH	KMeansL	LeeH	LeeL	VacH	VacL
1 thread	179±1.9	23.6±6.1	22.2±3.5	418±6.4	394± 4.0	524±9.4	475±3.7
2 threads	117±4.8	23.0±7.0	20.4±6.7	250±3.5	326±20.4	362±3.5	390±9.9
4 threads	88±2.9	21.3±3.4	17.7±3.3	162±2.1	280± 8.2	294±3.1	360±3.1
8 threads	83±9.1	27.6±7.8	16.4±4.5	115±4.1	285±14.2	291±3.0	381±9.7

Table 2. Average benchmark execution times in seconds, including one standard deviation, using the Priority contention manager.

is to see if the effect on performance varies with the CM used. Second, the remaining benchmarks are reported with all controller models, but only the Priority contention manager, as it is generally one of the better performing contentions managers. Using one of the better performing CMs will generally show the *minimum* benefit of using transactional concurrency tuning.

For each benchmark, dynamic execution should: (a) reduce execution time, over static execution with an initial number of threads that under-exploits the available parallelism, (b) reduce execution time, over static execution with an initial number of threads that over-exploits the available parallelism, and (c) reduce variance in execution time over different numbers of initial threads, compared to static execution time variance.

Figure 5 shows normalised execution time results for LeeH with all CMs. For static execution Aggressive, Kindergarten, and Priority CMs provide the best execution times, with a maximum difference of 3.2% between their respective best cases, and are the only CMs to show improving execution times up to 8 threads. The remaining CMs' performance degrades from 4 to 8 threads, indicating that either the proportion of time spent executing aborted transactions, or the time spent in resolving access conflicts, or both, has increased.

Intuitively, the 8 thread execution time improvements with dynamic execution for some CMs suggest that, at this number of threads, there are phases of execution where the available parallelism is low. With fewer threads, although such phases of low available parallelism may have occurred, they were not significant enough to cause a noticeable difference in execution time performance between static and dynamic execution.

Dynamic execution satisfies goal (a) above: execution time is always reduced when compared to static execution with 1 thread. Goal (b) is satisfied: dynamic execution improves execution time with 8 threads for five CMs. Goal (c) is also met: dynamic execution reduces variance in execution time compared to static execution, although execution time variance is not negligible for any controller model.

Additionally, for each CM (excluding Backoff), dynamic execution time for all numbers of initial threads is within 10% of the *best* static execution time. For Backoff this rises to 21% with EC. This shows that dynamic execution performance varies insignificantly with the CM used, including the best performing CMs (Aggressive, Kindergarten, and Priority). The results also show there is

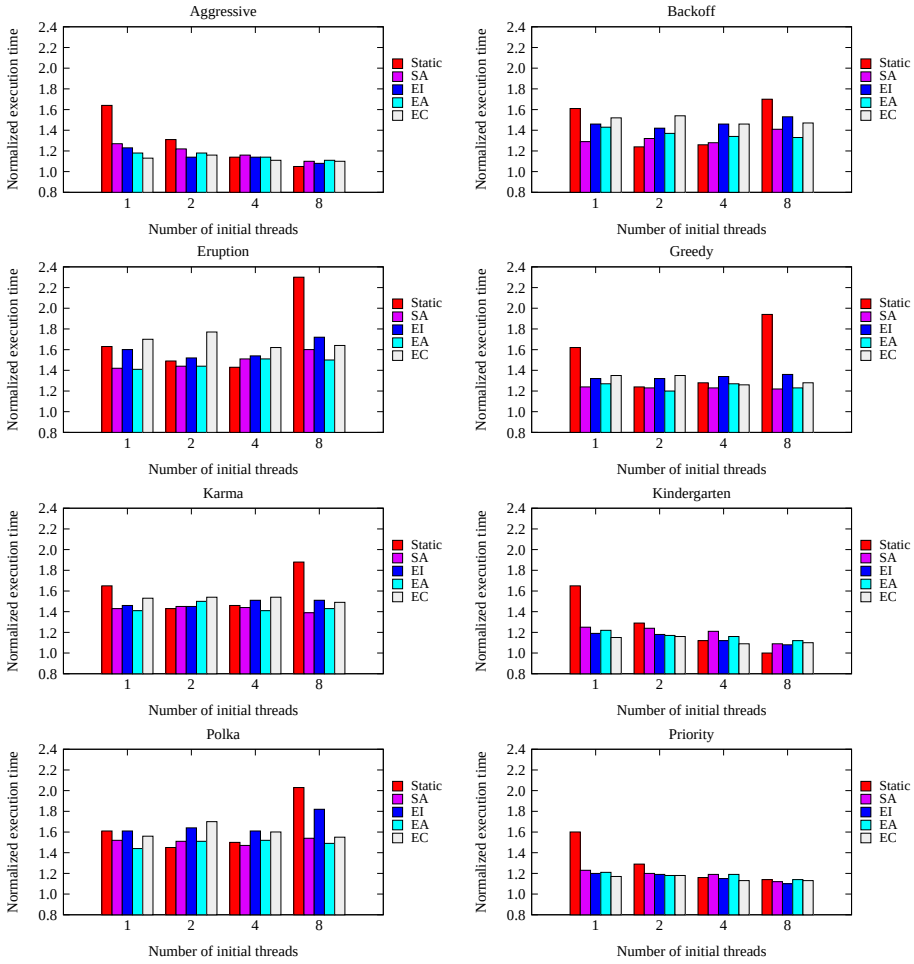

Fig. 5. Execution time for LeeH for each CM, normalised to overall best case static execution time (Aggressive with 8 threads). Less is better.

no clear winner amongst the controller models for any CM, but the variance amongst them is far smaller than amongst the CMs.

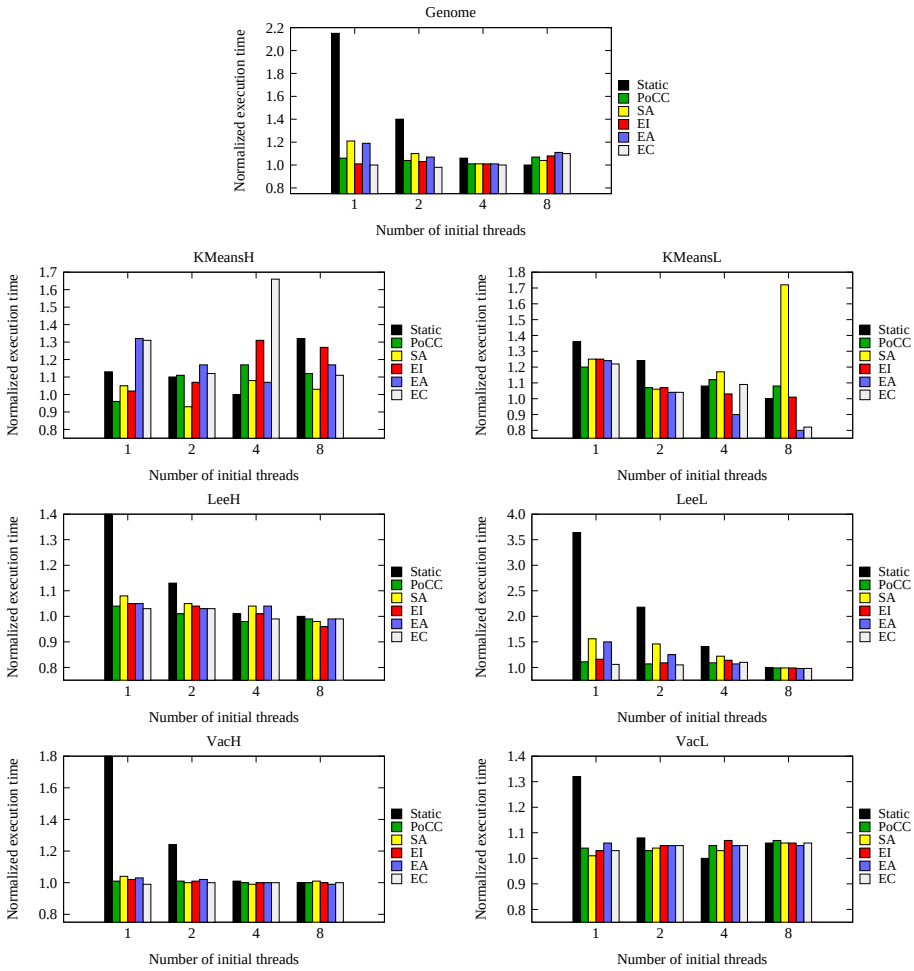

Fig. 6. Execution times for non-trivial benchmarks using the Priority CM, normalised to the best case static execution time for each benchmark configuration. StepChange benchmark omitted as it executes for a fixed duration. Less is better.

Figure 6 shows normalised execution time results for the benchmarks with only the Priority CM. Amongst these benchmarks only KMeansH and VacL do not improve execution time all the way up to 8 threads. KMeans experiments run for less than 20 seconds on average, thus the graphs have noise due to small execution time differences resulting in large variation in normalised execution time.

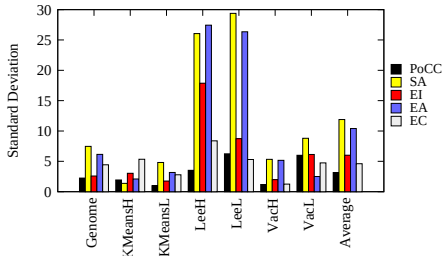

Fig. 7. Execution time std deviation over all initial threads. Less is better.

Looking at the 1 thread results, dynamic execution improves execution time results when static execution under-exploits available parallelism. In Genome and LeeL, EI and EC improve execution time better than SA and EA. Although EC does not fare well in KMeansH, the difference is not significant once execution time is taken into account.

Looking at results across all threads and all benchmarks, dynamic execution reduces variance in execution time results, with EI and EC showing less variance than SA and EA. Looking at the 8 thread results, dynamic execution only improves execution time when static execution over-exploits available parallelism in KMeans, because the falling execution times up to 8 threads show most of the benchmarks don't suffer from over-exploitation. Furthermore, the significance of the KMeans results is devalued by its short execution time.

Although the performance of the controller models with respect to best case static execution time is more variable in these experiments, the best case controller model for each benchmark degrades performance by 6% or less. Finally, again there is no clear winner amongst SA, EI, EA, and EC.

Generally, there is little difference in performance between PoCC and the first four controller models, but only PoCC consistently performs well across all benchmarks, whereas SA, EI, EA, and EC all show poor execution times in some benchmark configurations. Furthermore, averaging speedup of each controller model over static execution for each benchmark configuration, PoCC is second-best with an average speedup of 1.26, and EC is best with a marginally better speedup of 1.27. Averaging speedup of each controller model 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.

Figure 7 presents the execution time standard deviation for each benchmark to compare the effectiveness of the controller models at reducing execution time variance. The results show PoCC is the best on average, reducing standard deviation by 31% over the next best, EC.

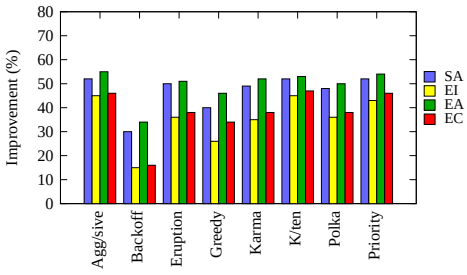

Fig. 8. Resource efficiency vs. static execution: 8 threads, LeeH.

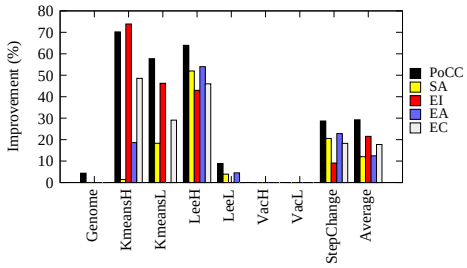

Fig. 9. Resource efficiency vs. static execution at 8 initial threads. More is better.

3.5 Resource Utilisation

Resource usage is calculated by summing, for all TCR samples, the sample duration multiplied by the number of threads executing during the sample. For each benchmark, dynamic execution should improve resource usage over static execution with an initial number of threads that over-exploits the available parallelism. Resource usage is compared for 8 initial threads, the system maximum, as applications that scale past 8 threads should show little resource usage improvement, and applications that do not scale past 8 threads should get maximum resource usage saving at 8 threads with dynamic execution, and thus allow comparison between the controller models. Again, the analysis is in two parts: LeeH with all CMs first, then all benchmarks with the Priority CM.

Figure 8 shows resource savings for all CMs with LeeH. Dynamic execution shows significant resource usage savings with many results in the 40-50% range. In particular, even the best performing CMs (Aggressive, Kindergarten, and Priority) have substantial resource savings, and, as presented earlier, dynamic execution still results in execution times that are similar to the best case static execution. The results also show that the relative savings between controller models is not affected by the CM used: EA always has the best savings, and EI the worst, for LeeH.

Figure 9 shows resource savings for all benchmarks, all controller models, with the Priority CM. Genome, Vac, and LeeL have little resource savings because they do not have low available parallelism at 8 threads. Relative savings are the same for LeeH and StepChange, but inverted for KMeans: EI offers larger resource savings amongst the first four controller models. However, PoCC is the best in every benchmark except KMeansH where EI is 3.7% better. On average, PoCC improves resource savings by 24% over the next best, EI.

3.6 Transaction Execution Metrics

Two transaction execution metrics are presented: *wasted work* and *aborts per commit* (APC), first presented in TM literature by Perfumo *et al.* [25]. 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 transactional concurrency tuning attempts to reduce variance in TCR, which should result in reduced variance in these metrics.

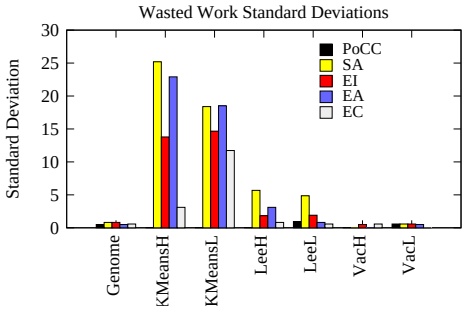

Fig. 10. Wasted work standard deviations for the benchmarks. Less is better.

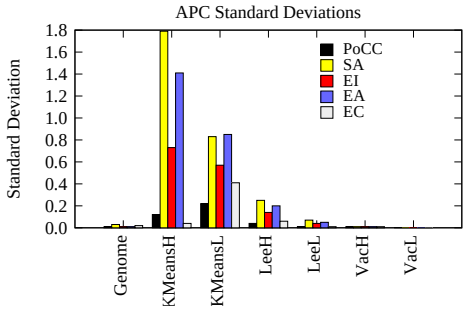

Fig. 11. APC standard deviations for the benchmarks. Less is better.

Figure 10 presents wasted work standard deviations. PoCC significantly reduces variability in wasted work: on average its standard deviation is 88% lower than the next best controller model, which is EC. Figure 11 presents APC, and again PoCC reduces variability: on average its APC standard deviation is 26% lower than the next best controller model, which is EC. Furthermore, PoCC reduces average wasted work by 16% over the next best controller model, which is EC, and reduces average APC by 11% over the next best controller model, which is also EC.

3.7 Response Characteristics

This section examines how the controller models respond to changes in TCR. Specifically, this section investigates how fast, how much, and how robustly, controller models respond. The responsiveness analysis is restricted to StepChange and LeeL. Both exhibit TCR profiles that stress the controller models as shown sampled at 1 second intervals in Figure 12. StepChange changes TCR by large amounts at fixed intervals, and LeeL has a wildly oscillating TCR due to the fast sample rate used to capture the data, but earlier sections have shown it has high available parallelism up to 8 threads.

Figure 13 shows the how the controller models respond to the changes in TCR. The first four controller models are robust to the noise in LeeL as the sample rate of the controller models is 10 seconds, not 1 second, which acts as a noise filter. The 50–80% *setPointRange* reduces the chance of unstable behaviour further.

However, these advantages turn into disadvantages for StepChange, where the first four controller models respond poorly. The *samplePeriod* gives the con-

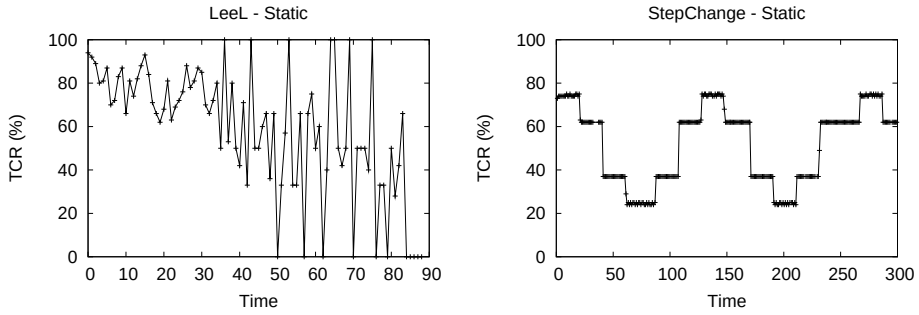

Fig. 12. TCR profiles of LeeL and StepChange, executing with 8 threads.

troller models response gradients that are not as steep as StepChange's changes in TCR. The *setPointRange* prevents the controller models from responding to smaller changes in TCR, despite all steps altering TCR by 15% or more. Finally, the *samplePeriodRange* used by EI, and EA, have an upper bound that is too high (60 seconds), resulting in EC failing to respond to the second trough in StepChange's TCR.

PoCC shows good response to both benchmarks: it is robust to noise in LeeL due to the *minTransactions* filter, and it responds to StepChange quickly due to the 1 second *samplePeriod*. The first four controller models trade robustness to noise for responsiveness by using a larger *samplePeriod*. PoCC removes the trade-off with its *minTransactions* filter, giving high responsiveness without compromising robustness.

4 Limitations

Transactional concurrency tuning has been shown to improve performance and reduce resource usage for a number of non-trivial benchmarks; in particular, PoCC has shown good response characteristics. Transactional concurrency tuning has also been implemented to support existing TM applications with only trivial changes. This section discusses two issues in relation to its widespread use: the practicalities of using transactional concurrency tuning, and the implications of using a thread pool to execute transactions.

The key to effective transactional concurrency tuning, as the earlier evaluation has shown, is the ability to quickly and adequately respond to changes in TCR. The system can only respond as fast as the selected *samplingPeriod*, and care needs to be taken to set this short enough to respond to the application's fluctuating TCR. However, if the TCR fluctuates at a rate near to, or faster than, the time it takes to execute the transactional concurrency tuning loop, then it is unlikely the system will be able to offer meaningful improvements in resource usage and performance. Hardware support for the control loop, for example maintaining the statistics needed to calculate TCR in hardware regis-

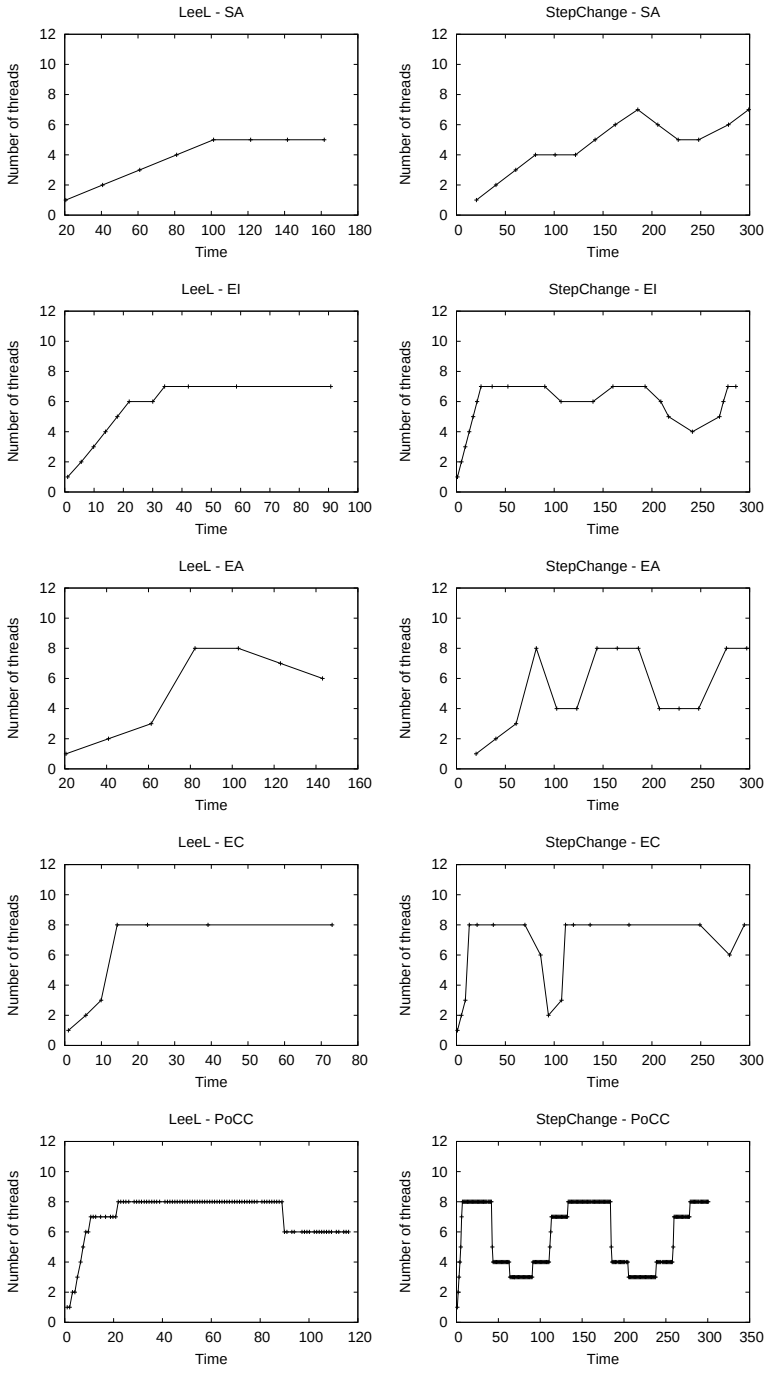

Fig. 13. Number of threads dynamically changing in response to changes in TCR using the Priority CM. All experiments start with 1 initial thread.

ters, may improve the loop's execution time, and improve the system's ability to support rapid TCR fluctuations.

A short *samplingPeriod* also adds overhead; the thread that executes the transactional concurrency loop code uses processor resources. However, with the increasing number of cores in multi-core processors, we do not foresee this to be an issue. Indeed, it may even be recommended to have the thread running continuously on its own core.

The thread pool is a different programming model from that seen in TM research, although it is not unfamiliar to the world of database transactions, on which TM is based. The thread pool has been refined to improve its scalability by implementing multiple work queues, and work stealing, and it is likely that further research in thread pools will continue to reduce their overhead as existing thread pool based applications move to multi-cores.

One issue is the increase in the total number of threads: application threads plus worker threads. Increasing the number of threads adds context switching overhead. However, it is likely that this overhead will be significantly reduced in multi-core architectures for two reasons. First, the increasing numbers of cores makes it natural to increase the total number of threads. Second, many multi-cores have added support for hardware context switching, which can switch thread contexts per processor clock cycle.

Other overheads that have not been investigated in this work include creation of data structures representing transactional jobs, job submission, and synchronisation when using synchronous job submission. Such overheads may be significant when executing very small transactions.

5 Conclusion

This paper has presented the first application of transactional concurrency tuning to TM with the aim of improving resource utilisation and execution time performance by adapting the number transactions executing concurrently to the available parallelism. A new metric, transaction commit rate (TCR), was introduced as a measure of available parallelism. Four transactional concurrency tuning algorithms (controller models) that varied in response strength were initially evaluated against a number of benchmarks and contention managers. The results showed transactional concurrency tuning led to execution time within 10% of the *best* non-transactional concurrency tuned execution time, whilst significantly reducing processor resource usage (over 40% in many cases) for those applications that exhibited phases of low available parallelism. The saved resources could be used by other applications, or powered down to save energy.

However, analysis of the controller models' response characteristics showed that they traded off robustness to noise in sampled TCR data, with responsiveness. This meant that the controller models' potentially needed their transactional concurrency tuning parameters re-tuning for every application they used, limiting their general applicability. A fifth transactional concurrency tuning algorithm, called PoCC, was created to address this problem, and incorporated

a relative gain formula and a minimum transaction count filter. Evaluation of PoCC showed it maintains average execution time similar to the best controller model, has the least performance deficit vs. best-case fixed-thread execution, and improves over the other four controller models by at least 24% average resource usage, 16% average wasted work, and 11% average APC. PoCC improves over the other four controller models 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 analysed. Finally, an analysis of all the controller models' 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. Kunle Olukotun and Lance Hammond. The future of microprocessors. *ACM Queue*, 3(7):26–29, September 2005.
2. Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *ASPLOS '96: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11. ACM Press, 1996.
3. Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, April 2005.
4. Richard McDougall. Extreme software scaling. *ACM Queue*, 3(7):36–46, 2005.
5. 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, May 1993.
6. 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.
7. James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool, 2006.
8. 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.
9. Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Lee-TM: A non-trivial benchmark for transactional memory. In *ICA3PP '08: Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing*. LNCS, Springer, June 2008.
10. 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.
11. Maurice Herlihy and Eric Koskinen. Checkpoints and continuations instead of nested transactions. In *TRANSACT '08: Third ACM SIGPLAN Workshop on Transactional Computing*, February 2008.

12. Torvald Riegel and Diogo Becker de Brum. Making object-based STM practical in unmanaged environments. In *TRANSACT '08: Third ACM SIGPLAN Workshop on Transactional Computing*, February 2008.
13. Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 185–196. ACM Press, February 2008.
14. Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216. ACM Press, February 2008.
15. Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 237–246. ACM Press, February 2008.
16. Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Investigating software transactional memory on clusters. In *IWJPDC '08: 10th International Workshop on Java and Components for Parallelism, Distribution and Concurrency*. IEEE Computer Society Press, April 2008.
17. 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.
18. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.
19. 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.
20. Karl Astrom and Tore Hagglund. *PID Controllers: Theory, Design, and Tuning*. Instrument Society of America, 1995.
21. William Scherer III and Michael L. Scott. Contention management in dynamic software transactional memory. In *CSJP '04: Workshop on Concurrency and Synchronization in Java Programs*, July 2004.
22. William Scherer III and Michael L. 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.
23. 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.
24. 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.
25. Cristian Perfumo, Nehir Sonmez, Adrian Cristal, Osman Unsal, Mateo Valero, and Tim Harris. Dissecting transactional executions in Haskell. In *TRANSACT '07: Second ACM SIGPLAN Workshop on Transactional Computing*, August 2007.