ADAPTIVE THREAD SCHEDULING TECHNIQUES FOR IMPROVING SCALABILITY OF SOFTWARE TRANSACTIONAL MEMORY

Kinson Chan, King Tin Lam, Cho-Li Wang
Department of Computer Science
The University of Hong Kong
Pokfulum Road, Hong Kong
{kchan, ktlam, clwang}@cs.hku.hk

ABSTRACT
Software transactional memory (STM) enhances both ease-of-use and concurrency, and is considered state-of-the-art for parallel applications to scale on modern multi-core hardware. However, there are certain situations where STM performs even worse than traditional locks. Upon hotspots where most threads contend over a few pieces of shared data, going transactional will result in excessive conflicts and aborts that adversely degrade performance. We present a new design of adaptive thread scheduler that manages concurrency when the system is about entering and leaving hotspots. The scheduler controls the number of threads spawning new transactions according to the live commit throughput. We implemented two feedback-control policies called Throttle and Probe to realize this adaptive scheduling. Performance evaluation with the STAMP benchmarks shows that enabling Throttle and Probe obtain best-case speedups of 87.5% and 108.7% respectively.

KEY WORDS
Software transactional memory, Adaptive concurrency control, Thread scheduling

1. Introduction

The rise of multicore processor architecture has reshaped supercomputing and marked the beginning of a new era. The shift to multi-core also marks an inflection point for mainstream software design philosophy [1]. Dissimilar readiness of software and hardware has presented an unprecedented challenge to software designers, preventing them from making best utilization of the rich hardware. To date, systems researchers are actively looking for promising parallel paradigms that permit future software both ease-of-use and high concurrency to solve this problem. Transactional memory (TM) has become a heavily reviewed candidate to increase the software-exposed parallelism for scaling with more and more cores. TM is a concurrency control mechanism analogous to database transactions for synchronizing access to shared memory among multiple threads. The first proposal [2] of using transactions as a consistency model dates back to 1993. Shavit and Touitou soon proposed the all-software approach to implementing TM in and coined the term software transactional memory (STM) [3]. STM is a speculative approach going for optimistic concurrency and relies on contention managers [4] to remedy conflicts. While attaining much better concurrency than lock-based synchronization, STM performance hinges heavily on the commit throughput all over the execution because every aborted transaction is simply wasting the CPU time. For applications that contain high contention by nature, STMs can perform even poorer than lock-based synchronization.

We argue that running parallel threads in a totally unsupervised but post-recovery manner via a contention management module is not the best but a "blind" STM infrastructure. When considering wider domains of parallel applications, STMs must be able to cope with the performance issue due to high contention. For virtues of high concurrency and high commit throughput to coexist at runtime, we propose new adaptive thread scheduling mechanisms to tune the level of concurrency dynamically, thereby reducing or even avoiding the chance of transactional conflicts when they are about to grow in number and aggravate the STM. On the other hand, if the thread scheduler observes that the present execution seldom encounters conflicts, it may speculate on boosting concurrency on-the-fly for better speedup. We see that research efforts [5, 6, 7, 8] in a similar direction yet nascent condition have lately begun (Section 5). To advance the state of the art, the contributions of this paper are two-fold:

1. By experimenting the STAMP benchmark suite [9], we conduct in-depth analysis (in Section 2) on the relation between the commit throughput and thread count, confirming that (a) variable concurrency is indeed wanted by different sections of the program execution; (b) the instantaneous commit rate or ratio is a trackable control parameter for dynamic tuning on the active thread count.
2. We develop an adaptive thread scheduler and design two concurrency control policies called Throttle and Probe (Section 3.2-3.3), pluggable to the scheduler. In particular, Probe is novel in its self-regulatory control logic against overreactions to commit ratio fluctuations. We implement our proposed techniques on TinySTM [10] (Section 3.4), and evaluate their performance with all STAMP applications (Section 4). The experimental results are generally positive and highlighted here: Throttle speeds up 6 out of 10 benchmarks with the best case up to 87.5%. Probe is effective to 4 applications (best-case gain is up to 108.7%; among the losing cases, three of them

suffer only slight slowdown within 10%). We also compare our solutions with the previous work done, namely Yoo's [6] and Shrink [7], whose codes have been ported to TinySTM. Averaging over all the STAMP benchmarks, we find that Throttle and Probe perform equally well as Yoo's and outperform Shrink by 13-14%.

2. The Excessive Threading Problem

In general, applications initiate an adequate number of threads to "exhaust" all the available parallelism. However, the inherent parallelism in the application may vary over the execution. We call the section of execution lacking inherent parallelism a *hotspot*. The more threads passing through a hotspot, the more likely transactional conflicts are seen. Analytic models made by Zilles, et al [11] show that conflict likelihood is proportional to c^2 where c denotes concurrency (i.e. concurrent transaction count). So running an application with a static thread count may see concurrency benefit for some duration but adverse performance effect around hotspots. In serious cases, excessive conflicts, aborts and retries due to hotspots could offset all benefits of the STM's optimistic approach.

We conducted an in-depth analysis for gaining insights into the problem of excessive threading, leading to ideas of our adaptive solutions. We run the STAMP benchmark suite [9] on an 8-core server supporting 16 hyper-threads (See Section 4.1 for detailed hardware configuration and brief characteristics of STAMP). We run the applications with initial thread count of 4, 8, 16 and 32. The 32-thread case is made deliberately for assessing extreme concurrency, emulating a more conflict-prone environment. Fig. 1 shows the variation of commit ratio against thread count. Commit ratio is the fraction of committed transactions out of all executed transactions. Practically, our commit ratios are taken per unit time. That means the system will reset the statistics of committed and executed transactions at regular time intervals. In theory, under low contention, performance gets improved by running more threads. This is because the transaction attempt rate increases with concurrency and the majority of attempts will become successful commits if contention is rare. On the contrary, running more threads under high contention worsens the commit rate and hence the performance. Our experimental result shown in Fig.1 confirms several hypotheses.

1) *Commit ratio truly varies along the execution*:
Nearly in all applications, the commit ratio keeps changing from time to time. Taking the 4-thread case of Intruder as an example, commit ratio keeps steadily to be over 85% during the initial stage but drops exponentially (implying more contentions) near the end of execution. This is normal if we cross-check the application nature. Intruder is a network intrusion detector. In order to detect intrusion attempts, the system first draws packets from a first-in-first-out queue. The worker threads try to link packets into network streams with a dictionary (a self-balancing tree), as well as retrieving completed streams for detection. There are more conflicts at the later stages because it takes more operations and time to access a tree than a queue. The conflict probability further increases when there are less data remaining in the tree. So commit ratio drops naturally. As a conclusive message to be taken by STM designers, there is no "cure-all" concurrency setting for applications and adaptive tuning is necessary to make the most of an STM system.

2) *Running fewer threads generally raises commit ratio*:
We can observe a common phenomenon from Fig.1 that except Kmeans-high and Bayes, all applications tend to attain a higher level of commit ratio when using fewer threads. Looking at Intruder again, commit ratio stays under 20% all the time at 32 threads but reaches 85% at 4 threads. Other applications show a similar behavior though the extent of variation differs. This leads to the idea that if the observed commit ratio stays poor, the system suspends some threads from running, thus helping the active ones commit more smoothly. Except those limited set of embarrassingly parallel applications, we expect most parallel applications running on shared-memory machines can benefit from such an adaptive control over the active thread count.

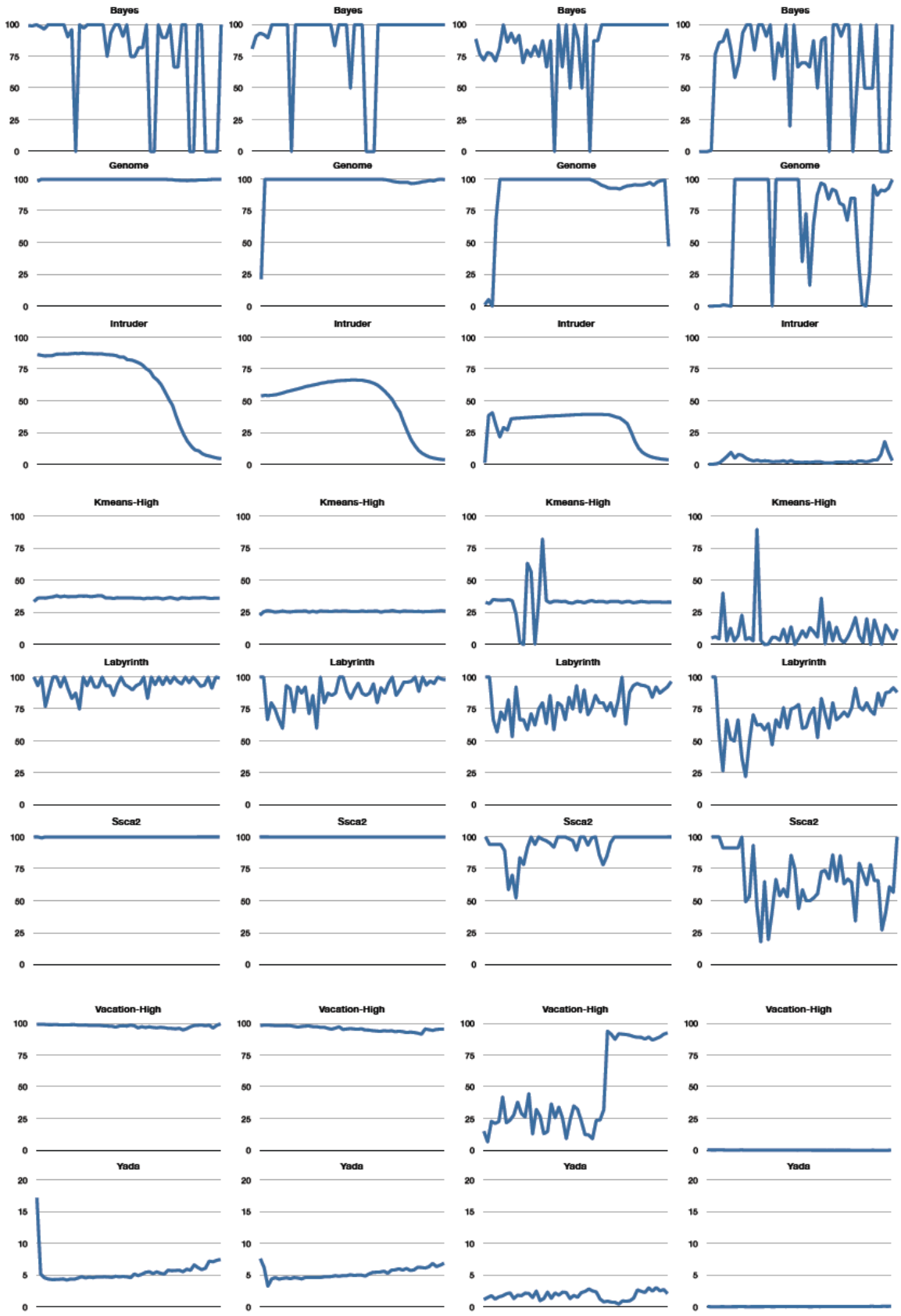
3) *Commit/abort statistics can serve as feedback control*:
If the commit ratio variation simply consists of random spikes and intense fluctuations all the time, it cannot be used as a control parameter for dynamic scheduling of active threads at all. Fig. 1 shows that the changing commit ratios in most applications indeed undergo traceable transitions. Exceptions are Labyrinth and 32-thread cases of Kmeans and Ssca2 where the commit ratio fluctuates promptly and are difficult to track. We can also see that scaling to 32 threads, the commit ratio curves tend to be more unstable. Fluctuation becoming more vigorous could serve as a sign of "over-threading" though our policies are not based on this property.

3. Adaptive Thread Scheduling Techniques

In this section, we present our adaptive concurrency control protocol. The essence of the protocol is similar to flow control of communication which aims not to overflow the medium: we don't want too many threads enter into concurrent transactions around the detected hotspots. The basic mechanism is based on using a dynamic quota parameter to limit active threads and hence concurrency. Different models or policies can be implemented to tune the quota. We develop two models called Throttle and Probe for doing so.

3.1 Quota-Driven Adaptive Concurrency Control

We introduce the following system parameters into the STM for concurrency control.
1. *quota*: number of concurrent threads allowed to enter into transactions, we call it the *concurrency quota*.



(a) 4 threads (b) 8 threads (c) 16 threads (d) 32 threads

Figure 1. Variation of commit ratio across execution time

2. *active*: number of active threads that have entered into transactions and not yet exited.
3. *peak*: peak number of threads that have begun transactions; including threads that have exited transaction.
4. *commits*: number of committed transactions.
5. *aborts*: number of aborted transactions.
6. *stalled*: a Boolean flag indicating that some threads are stalled at transactions' entrance for all concurrency quotas have been used up, i.e. *active* has equaled *quota*.

Among them, *quota* is the tuning target; *commits* and *aborts* are observed statistics channeled from the STM runtime; *active*, *peak* and *stalled* are derived variables facilitating the control algorithm. Note that variables 3-5 are rate-based: we accumulate and reset their values per regular time intervals, e.g. 5 ms. So *commits* means the number of committed transactions within the current interval, not the total accumulated since the program start. We would write *last_commits* in later context to mean the count of committed transactions within the last interval.

Table 1 shows the basic mechanism to operate on the concurrency quota concept. When a thread tries to begin a new transaction (i.e. call the STM's `begin()` function), it has to check if the current active thread count has reached the current quota allowed. If this is the case, it has to wait in place until either some active threads get exited transactions (in other words, some transactions get committed or aborted) or the current quota is lifted up on demand by the background daemon running some tuning policies to be explained in Sections 3.2 and 3.3. We added an extra heuristic to the mechanism: we record the active thread count that once happens to be the highest into the *peak* variable. If the daemon finds that the current quota is even larger than peak, it will set quota to peak. This measure is to avoid quota being incremented beyond the highest concurrency need. Superfluous quota will simply oppose the virtue of putting a bound on concurrent thread count.

Table 1. Concurrency Quota Mechanisms

```
function onBegin
retry:
  if active >= quota then
    set stalled to true
    yield
    goto retry
  end if
  active := active + 1
  if peak < active then
    peak := active
  end if
end
```

```
function onCommit
  active := active - 1
  commits := commits + 1
end

function onAbort
  active := active - 1
  aborts := aborts + 1
end
```

Table 2. Throttle Policy

```
while true do
    sleep for a constant time (e.g. 5ms)
    if commits + aborts < warmup then
        continue
    end if
    ratio = commits / (commits + aborts)
    if peak < quota then
        quota := peak
    else if ratio < threshold then
        quota := quota - 1
    else if stalled is true and ratio > threshold then
        quota := quota + 1
    end if
    reset peak, commits, aborts to zero and
    stalled to false
end do
```

Note: Since *peak* and other counters will be reset to zero across regular intervals, e.g. per 5 ms, when the daemon wakes up, the heuristic of setting *quota* to *peak* if *quota* > *peak* will set the current quota to zero, making all threads stalled! So there is a protective measure to forbid the daemon from updating the quota until *commits* + *aborts* >= *warmup* (a predefined constant, e.g. 10). At the moment that *commits* + *aborts* observed in the current interval get larger than 10, there must have been 10 threads called `onBegin()`, so *peak* should be non-zero right now. Then *quota* will never vanish to halt the entire system.

3.2 Throttle Policy

Table 2 shows the algorithm of our first feedback control policy. The policy name is coined by analogy with a car's throttle that regulates airflow into the engine and hence controls how fast the engine goes. A driver normally impedes the car if seeing many cars or even accidents on the road. Likewise, we want the STM to suppress concurrency if it sees abort count to rise. Quota increment and decrement are analogous to pressing and releasing the gas pedal by the driver. Since *aborts* would rise when some threads are entering a shared memory hotspot, the observed commit ratio, (denoted by *ratio* in Table 2) would drop. If the ratio drops below some predefined *threshold*, say 0.8, the system would narrow the quota, analogous to a driver releasing the gas pedal. If the hotspot lasts for a sufficient period that allows threads to see the new quota, commit ratio has good chance to rise. Upon leaving a hotspot, aborts drop and commit ratio gets back to rise beyond the threshold, quota is relaxed (when seeing *stalled* = true) for resuming stalled threads to drive parallel execution at full throttle.

3.3 Probe Policy

We show another control policy design in Table 3. The name Probe implies the algorithm is probing for a concurrency quota that optimizes *commits* by some "trial and error" method. In theory, there exists an optimal of active thread count that corresponds to the crest of the commit rate curve which looks bell-shaped. Note that in this policy, we use *commit rate*, i.e. number of committed transactions per unit time, rather than commit ratio. We quantify the unit time logically by *laps*. Similar to Throttle, this policy employs a protective postpone of quota update until *commits* + *aborts* >= *warmup*. What is extra is we add a counter *laps* to record the count of daemon sleeping cycles that have passed until warm-up is done. So commit rate can be calculated as *commits* divided by *laps*.

Table 3. Probe Policy

```
set direction to down
while true do
    sleep for a constant time (e.g. 5ms)
    if peak = 0 and active = 0 then
        continue
    else if commits + aborts < warmup then
        laps := laps + 1
        continue
    else
        laps := laps + 1
    end if
    if peak < quota then
        quota := peak + 1
        set direction to down
    else if quota = 1 then
        set direction to up
     else if commits / laps < last_commits / last_laps then
        set direction to reverse(direction)
    end if
    if direction is down then
        quota := quota - 1
    else
        quota := quota + 1
    end if
    last_commits := commits
    last_laps := laps
    reset peak, commits, aborts, laps to zero
end do
```

Table 4. Qualitative Summary of the Stamp Benchmark

App	Tx Length	R/W Set	Tx Time	Contention
bayes	Long	Large	High	High
genome	Medium	Medium	High	Low
intruder	Short	Medium	Medium	High
kmeans	Short	Small	Low	Low
labyrinth	Long	Large	High	High
ssca2	Short	Small	Low	Low
vacation	Medium	Medium	High	Low/Medium
yada	Long	Large	High	Medium

In reality, the optimal point for the current active thread count is floating: it may keep shifting horizontally along the execution time. We could make the system stay close to the sweet spot continuously by a probing technique as follows. If we use fewer threads but observe a drop in commit rate (refer to the check: if *commits* / *laps* < *last_commits* / *last_laps* in Table 3), this implies we are falling down hill and getting further away from the optimal. Thus we should reverse the tuning direction from down to up, i.e. keep incrementing the quota until we start to see another drop of commit rate, which implies "over-relaxation" on the quota.

3.4 System Implementation

We implemented all the abovementioned in TinySTM v.0.9.5 [10]. We pick this specific version in order to have a fair comparison with prior related work on concurrency control policies, Yoo's [6] and Shrink [7], that have gone open-source and bundled in this TinySTM version.

Thread stalling at lacking quota and periodic daemon wakeups are realized by spins over `sched_yield` and `usleep` system calls in Linux. The variables *active*, *quota* and *peak* are stored as bits of a 64-bit integer and modified via compare-and-swap (CAS) operations in an obstruction-free [12] mode, i.e. a preempted thread will not block other threads from starting or finishing transactions.

4. Performance Evaluation

4.1 Evaluation Platform and Methodology

Our experiments were conducted on a multicore server of our PC cluster [13]. The server hardware configuration is as follows: 2 × Intel E5540 (Nehalem-based) Quad-core Xeon 2.53GHz CPUs (i.e. total 8 cores), 32 GB 1066MHz DDR3 RAM and SAS disks/RAID-1. Both CPUs have hyper-threading enabled, supporting concurrent run of 16 threads. The operating system is Fedora Core 11.

We evaluate our solutions using the STAMP benchmark suite with modifications tailored for TinySTM. Table 4 comes from the original STAMP paper [9]. Columns 2 to 5 represent length of transactions (number of instructions), size of read and write sets, portion of time spent on transactions, and amount of contention respectively. For comparison purpose, we obtained implementation of Shrink and Yoo's policies, tied with TinySTM v.0.9.5, from the website of Distributed Programming Laboratory of EPFL [14]. We ran the 10 test cases unmodified to see how much speedup the four concurrency control policies namely, Throttle, Probe, Shrink and Yoo's can gain over the plain execution without concurrency control. For conciseness, we use the abbreviation ACC (adaptive concurrency control) in later context to collectively refer to any of these policies. Some applications show high discrepancies in execution times across runs. We handled this by repeating each test for 7 times and taking the average.

4.2 Experimental Results

Fig. 2 shows the scalability of TinySTM, with and without ACC. Each separate chart corresponds to one application (kmeans and vacation have two test cases: low and high in terms of inherent contention). The x-axis represents the static initial thread count (ITC) spawned by the benchmark. Except "original" (i.e. no ACCs), the actual number of active threads could be different from time to time (but never > ITC).

TinySTM shows increasing speedup for ITC between 2 to 8. Speedup drops when ITC reaches 16 or 32, depending on the application nature. In general, for a machine of 16 hyper-threads, one cannot expect any speedup to gain by using thread count beyond 16, so the 32-thread data point is just for reference, showing how the STM could behave under extreme condition. We can see in some cases (mostly the "original" curve), speedup drops below one when scaling towards 32 threads (an exceptional case is bayes where speedup keeps further increas-

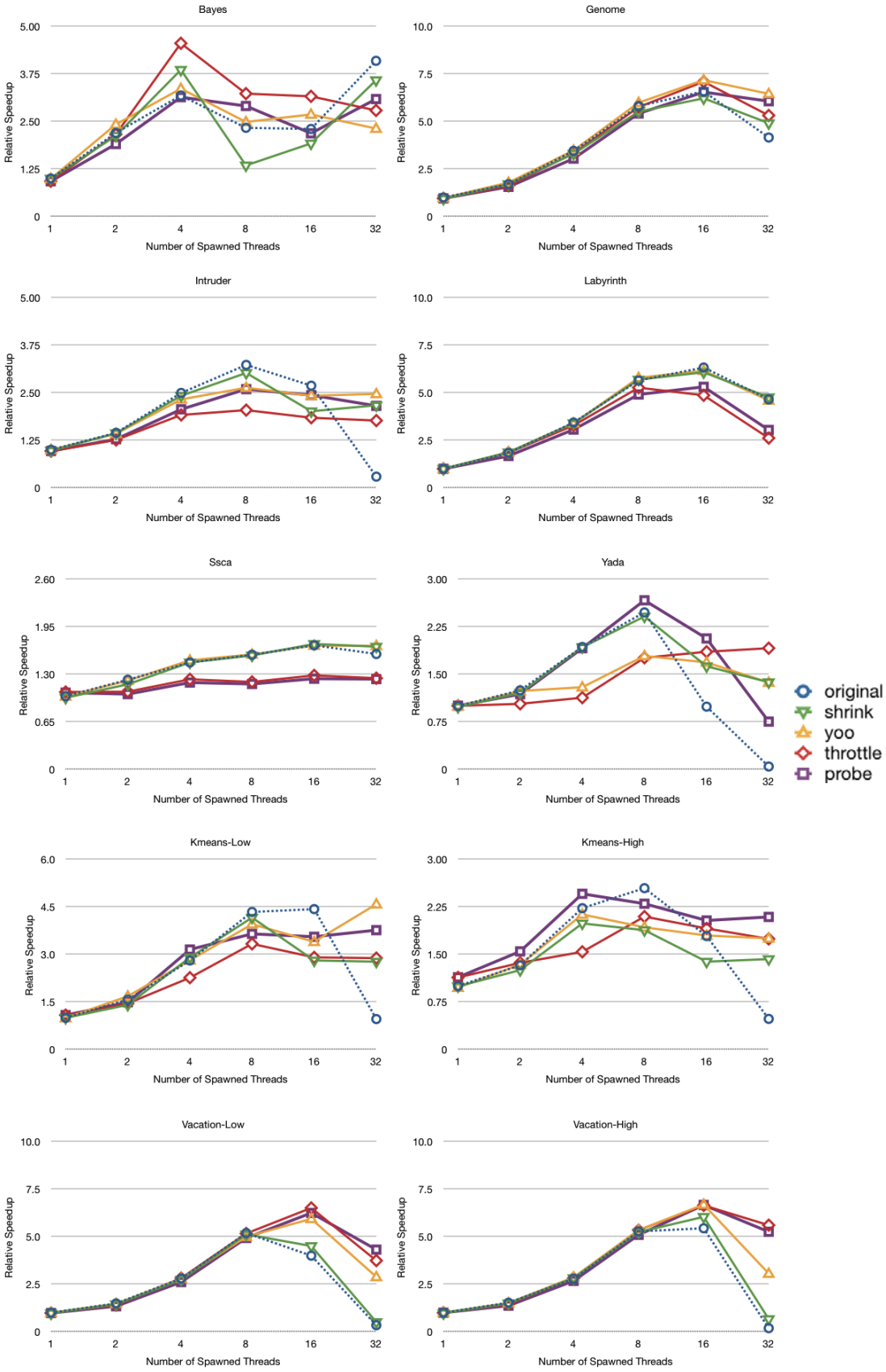
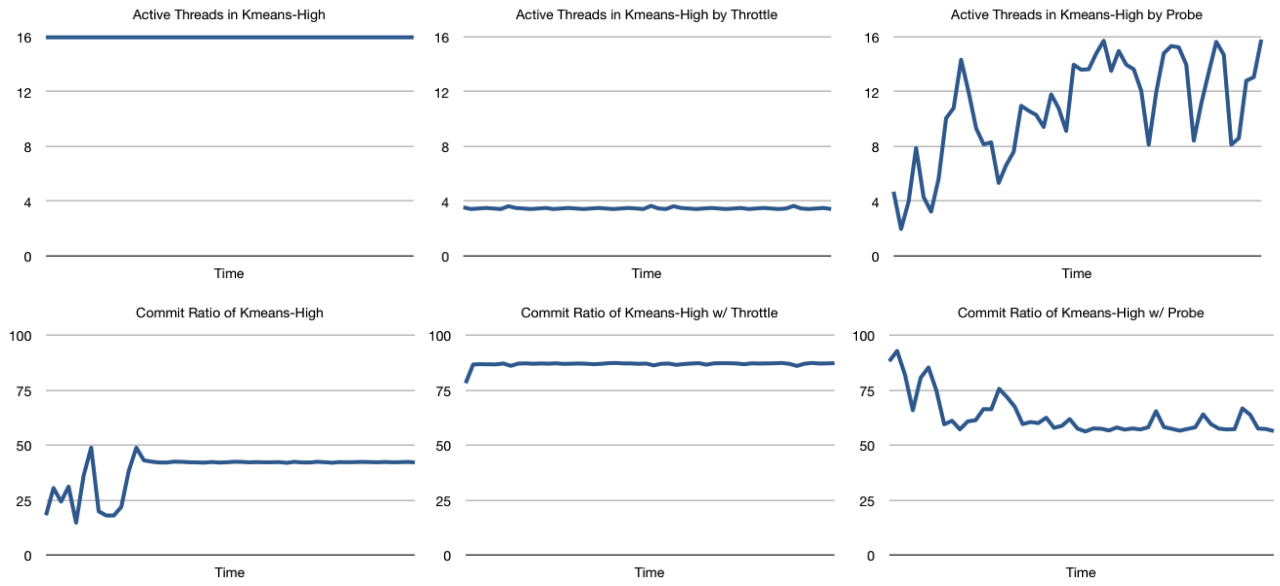


Figure 2. Speedups obtained with different concurrency control heuristics



(a) Without concurrency control (b) With Throttle (c) With Probe

Figure 3. Variation of active thread count and commit ratios in kmeans-high across execution time

Table 5. Comparison of ACC Policies

Policy	Best Case	Worst Case	Average
Shrink	+64.32% (yada)	-36.60% (kmeans-1ow)	-2.1%
Yoo	+70.50% (yada)	-23.27% (kmeans-low)	+13.2%
Throttle	+87.49% (yada)	-34.39% (kmeans-low)	+11.1%
Probe	+108.69% (yada)	-26.94% (ssca)	+12.4%

ing). The reason is contention gets severer along increasing threads. When ACC policies are enabled, we can perceive their rectifying effect that speedup drops less rapidly (some of the downhill curves become even bent upward; e.g. Throttle for yada). Meanwhile, we notice in some benchmarks (kmeans-low) that enabling ACCs could reduce speedup or scalability even for ITC within normal range (2-8). Although some programs get slowdown from ACCs enabled, ACCs show positive effect from an average point of view. Table 5 shows the best-, worst- and average-case gain in speedup of the various ACC policies compared to plain execution without ACC. The average is taken over all 10 test cases. We see an overall improvement of 11-13% by enabling an ACC protocol (except Shrink). Throttle and Probe improve on the best-case gain and outweigh Shrink and Yoo's. In particular, Throttle outpaces others for bayes and vacation, while Probe performs best for yada, kmeans-high and vacation-low/high. Overall, Throttle speeds up 6 out of 10 benchmarks with the best case up to 87.5% (yada). Probe is effective to 4 applications with the best-case gain up to 108.7%, and among the losing cases, 3 of them suffer only slight slowdown (< 10%). On average (considering all 10 benchmarking test cases), Throttle and Probe perform equally well as Yoo's but outrun Shrink by 13-14%.

We see that kmeans-low is defying all ACC protocols and makes the worst case. The reason behind is that it involves too frequent transactions of short lengths. Since the ACC mechanisms instrument the `begin` and `commit` procedures, a slight addition of overhead will be amplified as drastic performance degradation. Moreover, as there is not much contention, the benefit of having concurrency control cannot cover the overhead. For kmeans-high whose scalability is per se low, the overhead of concurrency control is shadowed by its benefits. We also try to link the performance benefits in kmans-high to the improved commit ratios after applying Throttle and Probe heuristics. Fig. 3 shows the variation of active thread count over execution time and its corresponding effect on the commit ratio in kmeans-high. Fig. 3(a) depicts the inherent commit ratio behavior of the application. Fig. 3(b) shows that Throttle bounds the active thread count to 4, effectively raising the commit ratio to a level of about 85%. However, this heuristic may be overly strict on concurrency, i.e. overreaction. On the other hand, Probe follows a pendulum-like model to approach the current optimal active thread count rather than to purely sacrifice concurrency for a prolonged period as in Throttle. So we can see more variations in Fig. 3(c). Reconcile with Fig. 2, we can see this strategy allows Probe to gain more speedup, winning over Throttle and all prior heuristics.

Yada is an application with commit ratio well below 20% when it is run with TinySTM. Both Throttle and Yoo policy are misled by frequent aborts to reduce number of threads. Shrink, which performs hotspot detection, reduces the amount of unnecessary serialization and performs generally better than Yoo's. Our Probe policy actively searches for maximum value of commit rate and successfully does so when there are less than 16 threads. When there are 32 threads, however, the variance of commit rate fluctuates a lot and causes the heuristic to fall.

Vacation has a random data access pattern. Yoo's and our policies handle threads uniformly and perform well. On excessive threads, performance drops rapidly as the chance of conflict undergoes quadratic growth. Our policies defer execution of some threads, successfully reduce conflicts and have shorter program completion time.

5. Related Work

Ansari et al [5] proposed control mechanisms to adjust active thread count according to the so-called transaction commit rate (TCR) which we would regard as commit ratio. They tested only with sparingly few applications to show the effectiveness. Our experimental evaluation on Throttle with diverse benchmarks fills this gap. Our new policy Probe is more immune to overreacting behavior which is a consequence of observing commit ratio.

Yoo, et al [6] proposed another mechanism to adjust active thread count. Unlike Ansari's work, Yoo's policy does not require heuristic data sharing among threads. It computes contention intensity (CI) on each thread. When CI is above a threshold, the thread acquires a common lock before starting a new transaction. However, counting the raw conflict count is unreliable since large amount of conflicts (e.g. in Yada) may mislead the policy to unnecessarily serialize the transactions. Our Probe policy disregards the rollback count and is freed from this problem.

Shrink [7] assumes conflicts are induced by memory hotspots. It adaptively activates hotspot detection when a thread encounters repeated conflicts. Transactions that are known making access to hotspots are required to acquire a common lock so that they serialize among themselves without affecting other threads. Unfortunately, if data access of a program is purely random, all these efforts do nothing helpful but waste even more computing time, as shown in vacation, where Shrink does not significantly bring improvement. In contrast, our heuristics handle this case properly by reducing active threads system-wide.

CAR-STM [8] assumes some pairs of threads share data all the time. It features a scheduling-based mechanism for collision avoidance and resolution. It schedules transactions that are likely to conflict to the same processor, effectively serializes them and reduces number of conflicts. While we have not evaluated this system, we think it may suffer from the same problem of Shrink that some threads are unnecessarily serialized in applications with random access patterns.

6. Conclusion

This paper demonstrates two adaptive concurrency control techniques called Throttle and Probe that can effectively avoid negative effect of excessive threading and bring about performance gain. Throttle aims for high commit ratio while scheduling. Probe aims for seeking the sweet spot on commit rate by varying number of active threads, and is more robust in that it does not get misled by low commit ratios. Our results show that Probe outperforms existing concurrency control protocols in applications where the commit ratio is low. In future we may consider combining the two policies to bring performance improvement to a broader set of applications. We will also investigate the notion of adaptively selecting different adaptive concurrency control and STM protocol parameters according to live commit statistics.

Acknowledgement

This research is supported by Hong Kong RGC Grant HKU7179/09E and Hong Kong UGC Special Equipment Grant SEG HKU09.

References

[1] Maureen O'Gara. The Intel roadmap: The shift to multicore is an inflection point for software design philosophy. 2005. http://opensource.sys-con.com/node/48477.

[2] M. Herlihy, and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[3] N. Shavit, and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, 1995

[4] W. N. Scherer III, and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 240–248, 2005

[5] M. Ansari, C. Kotselidis, K. Jarvis, M. Lujan, and I. Watson. Advanced concurrency control for transactional memory using transaction commit rate. In the *Proceedings of the 14th International Euro-Par Conference on Parallel Computing*, pages 719–728, 2008.

[6] R. M. Yoo, and H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In the *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 169–178, 2009.

[7] Dragojevic, A. V. Singh, R. Guerraoui and V. Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 7–16, 2009.

[8] S. Dolev, D. Hendler, and A. Suissa. CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 125–134, 2008.

[9] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of The 2008 IEEE International Symposium on Workload Characterization*, pages 35–46, 2008.

[10] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 237–246, 2008.

[11] C. Zilles, and R. Rajwar. Implications of false conflict rate trends for robust software transactional memory. In *Proceedings of the 10th International Symposium on Workload Characterization*, pages 15–24, 2007.

[12] M. Herlihy, V. Luchangco, and M. Moir. Obstruction free synchronization: Double ended queues as an example. In the *Proceedings of the 32rd International Conference on Distributed Computing Systems*, pages 522–529, 2003.

[13] HKUCS SRG. HKU Gideon-II Cluster. http://i.cs.hku.hk/~clwang/Gideon-II/.

[14] LPD-EPFL. How good is a transactional memory implementation. http://lpd.epfl.ch/site/research/tmeval.