# Cache Affinity Optimization Techniques for Scaling Software Transactional Memory Systems on Multi-CMP Architectures

Kinson Chan
The University of Hong Kong
Hong Kong, China
Email: kchan@cs.hku.hk

King Tin Lam
The University of Hong Kong
Hong Kong, China
Email: ktlam@cs.hku.hk

Cho-Li Wang
The University of Hong Kong
Hong Kong, China
Email: clwang@cs.hku.hk

***Abstract*—Software transactional memory (STM) enhances both ease-of-use and concurrency, and is considered one of the next-generation paradigms for parallel programming. Application programs may see hotspots where data conflicts are intensive and seriously degrade the performance. So advanced STM systems employ dynamic concurrency control techniques to curb the conflict rate through properly throttling the rate of spawning transactions. High-end computers may have two or more multi-core processors so that data sharing among cores goes through a non-uniform cache memory hierarchy. This poses challenges to concurrency control designs as improper metadata placement and sharing will introduce scalability issues to the system. Poor thread-to-core mappings that induce excessive cache invalidation are also detrimental to the overall performance. In this paper, we share our experience in designing and implementing a new dynamic concurrency controller for TinySTM, which helps keeping the system concurrency at a near-optimal level. By decoupling unfavorable metadata sharing, our controller design avoids costly inter-processor communications. It also features an affinity-aware thread migration technique that fine-tunes thread placements by observing inter-thread transactional conflicts. We evaluate our implementation using the STAMP benchmark suite and show that the controller can bring around 21% average speedup over the baseline execution.**

***Keywords*—*Software transactional memory, concurrency control, thread migration, multicore processors, cache affinity***

## I. Introduction

During the last decade, we have witnessed the dominance of multicore processors (a.k.a. chip multiprocessors, CMPs) and their success in fueling the next computing performance leap beyond the ultimate limit of CPU clock rate scaling. A multi-level cache is one of the most important resources of a CMP.[1] Some CMP designs prefer keeping the last-level cache private to each core for simplicity, while other architectures expose it for sharing among different cores for faster communication and improved resource utilization. To unleash extreme parallel performance, it is common for high-end machines to have two or more CMPs interconnected through high-speed links such as Intel QuickPath Interconnect and AMD HyperTransport. Such *multi-CMP systems* present both optimization chances and challenges to system architects in regard to data sharing between threads along the hierarchy of private, shared and remote caches.

Considering the significant gap between intra- and inter-CMP access latencies, incorporating cache affinity techniques into software systems design is an inexorable step to harness the full power of multi-CMP architectures. Apart from this, a longstanding barrier to fully unlocking such power is rooted in traditional lock-based synchronization that creates serialized sections of execution limiting the available parallelism. This calls for a radical change in the parallel programming model. *Software transactional memory (STM)*, being the most heavily purported programming paradigm to replace locks for multi-core systems, is undoubtedly a kind of system software that needs to apply the cache affinity principles. Access conflict in a transaction is a good indicator of data sharing between a pair of threads. By collecting sufficient contention statistics, the STM may depict the inter-thread sharing patterns, and can make use of the CPU affinity mask to reconfigure thread-to-core mappings and pin down favorable ones. Both the application code and the STM system activities such as conflict resolution will see enhanced cache locality when shared data stay in high-speed caches most of the time. Besides application data, the STM itself has extra internal data structures for keeping metadata. When implementing an STM, care must be taken to avoid penalizing the overall system execution speed by adding code that entails intensive metadata updates going through inter-CMP links.

The STM paradigm has many potential merits like much higher concurrency than locks. But for high-contention applications, STMs can perform even worse than locks due to too many access conflicts, rollbacks and retries. For high concurrency and high commit throughput to coexist at runtime, we cling to the notion of *supervised concurrency level*. With adaptive thread scheduling mechanisms, the STM runtime tunes the level of concurrency dynamically. The scheduler may suspend some active threads in place to let others proceed their commit procedures smoothly, thereby minimizing the chance of access conflicts when they are to grow in number. On the other hand, if the scheduler observes that the present execution seldom encounters conflicts, it may raise the concurrency level on the fly to approach the full application parallelism.

There have been several research efforts on adaptive concurrency control [1], [2], [3], [4] but we observe two areas

[1] In this paper, the term *CMP* or *processor* refers to a chip package (installed in a CPU socket) that contains multiple cores and a shared cache.

falling short thus far. First, the existing feedback-controlled concurrency tuning mechanisms and policies are rather simple. They could be misled by the observed commit ratios, resulting in overly strict concurrency control—the worst case could be a slowdown of nearly 30%. Second, they were not carefully designed with respect to the multicore cache hierarchy, especially on a multi-CMP system.

In this paper, we demonstrate an advanced word-based STM combining the concepts of supervised concurrency and cache affinity to boost the overall system performance. Based on our previous work, we develop an enhanced version of the *Probe* concurrency control protocol [5] for tuning the runtime concurrency with low overheads. In terms of novel technical contributions, we propose two cache-affinity techniques, namely *metadata zoning* and *affinity-driven thread migration*, to enhance the performance of the Probe-controlled system. Metadata zoning aims to confine metadata sharing to within a CMP rather than going through the inter-CMP paths. Thread migration aims to reconfigure thread-to-core mappings to improve cache affinity. We implement the methodology into the TinySTM library and evaluate its performance. Experimental results show that the enhanced system can obtain performance improvement of about 100% (best-case) and 21% (average over all benchmarks) compared with the baseline execution without concurrency control and also outstrip previous work like Shrink [1] as well as Yoo and Lee's ATS [3] significantly.

For the rest of the paper, we will give the background information on modern cache hierarchy, STM basics and the relationship between commit throughput and active thread count in Section II. Section III presents the Probe concurrency control enhanced with the metadata zoning technique. Section IV discusses the affinity-driven thread migration algorithm. We explain our system implementation in Section V and evaluate the performance in Section VI. Related work is reviewed in Section VII. Finally, Section VIII concludes this paper.

## II. Background

In this section we discuss multicore (CMP) processors and software transactional memory systems. We highlight the difficulties in having a precise concurrency control in STM, especially on multi-CMP systems.

### A. Modern Computer Cache Hierarchy

Most modern computers employ a multicore processor (a.k.a. chip multiprocessor, CMP). A multicore processor is a single chip containing two or more cores, each of which can run at least one hardware thread.[2] Threads within the same core share the L1 cache while threads on different cores of the same CMP share L2 or L3 caches.

To pursue high performance, high-end computers can have multiple multicore processors, complicating the cache sharing performance. Threads on different processors do not share any common cache. Fig. 1 depicts the organization of a common two-way multicore multiprocessor (or multi-CMP) computer. In the example, we see that thread 1 and thread 3 share the same L3 cache for fast data sharing, whereas thread 3 and thread 5 share data through the narrower and longer inter-processor communication channel.

[2]State-of-the-art CPUs leverage chip multithreading technology to enable two or more threads to run on each core

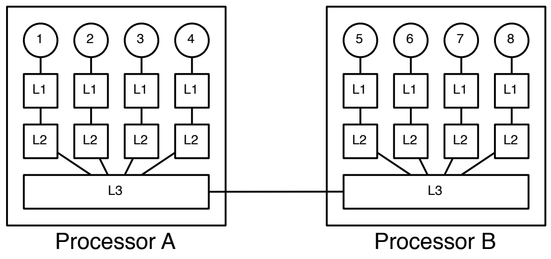

Fig. 1. An example organization of a two-way multicore computer

TABLE I. Ping-Pong Rates of Different Intel Multicore Processors

| Processor | Clockspeed | Intra-Proc | Inter-Proc |
|---|---|---|---|
| Core 2 Quad Q6600[1] | 2.40 GHz | $1.6 \times 10^7$ | $3.9 \times 10^6$ |
| Xeon E5540 | 2.53 GHz | $1.3 \times 10^7$ | $5.4 \times 10^6$ |
| Xeon E5550 | 2.66 GHz | $1.5 \times 10^7$ | $6.3 \times 10^6$ |
| Xeon X5670 | 2.93 GHz | $1.4 \times 10^7$ | $6.2 \times 10^6$ |

[1] Although Q6600 comes in a single quad-core package, it features two sets of shared L2 cache, each to be shared by two cores only. By our definition in Section I, we treat it as two *processors*.

To characterize intra-processor and inter-processor communication speeds, we run ping-pong tests on several systems equipped with common multicore processors. Table I shows the results (the last two columns show the number of round-trip transfers of a cache line between two cores). We notice that the inter-CMP communication cost is up to a factor of 4 higher than the intra-CMP one.

### B. Software Transactional Memory

Software transactional memory is an optimistic approach of parallel programming. Programmers mark critical sections as *transactions* and the STM per se will ensure that each of the transactions is executed atomically. STM speculatively runs the transactions in parallel and checks for conflicts (i.e. violation to the atomic property). Some transactions involved in a conflict will be aborted (with their updates rolled back) during conflict resolution. When a transaction reaches to the end without encountering any conflict, it commits and makes the updates visible to upcoming transactions. STMs provide interfaces `stm_begin()` and `stm_commit()` for purpose of starting and finishing a transaction.[3] When a conflict is detected, the system automatically aborts at least one of the transactions by calling `stm_abort()`, through which each of the victim threads is reset to where `stm_begin()` was called and all its speculative updates are discarded.

STMs can be classified in a few dimensions. We can classify the systems according to progress guarantee and the granularity of data sharing. There are several common progress guarantees—lock-free, obstruction-free [6] and blocking. Lock-free systems guarantee that at least one of the transactions can proceed when multiple transactions run into conflicts. Obstruction-free systems only guarantee that a transaction can be eventually successful at repeated retries. While there exists an obstruction-free word-based STM contributed by Marathe et al. [7], lock- or obstruction-freedom makes

[3]These functions are common among most STMs but their names may vary.

efficient implementation typically difficult. Inspired by Ennals' work [8], blocking (or lock-based) STMs are recently under the research spotlight as they can simplify the design and provide higher performance. As a result, most of the latest word-based STM implementations such as TL2 [9], TinySTM [10] and SwissTM [11] are blocking.

Blocking STMs do not have progress guarantee. Doomed transactions (transactions that will eventually roll back) may hold ownership of some data items, obstructing the progress of other transactions. In these systems, it becomes more important to control the concurrency since excessive threading is just wasting processor time and hurting system performance.

## C. Dynamic Concurrency Control for STM

Transactional memory helps tapping into the maximal parallelism of a program while maintaining the ease of programming similar to coarse-grained locking. Performance of software transactional memory can be quantitatively represented by the commit rate, which can be further identified as a product of the transaction attempt rate and commit ratio (#committed transactions divided by #attempted transactions).

Transaction attempt rate is dependent on various conditions. First of all, it is related to the number of active threads. When there are more threads, there are more concurrent transaction attempts. It is also related to the commit ratio—there are more retry attempts when there are more rollbacks. The attempt rate is affected by the transaction length as well. Longer transactions lead to fewer attempts in a unit time as each of them takes longer time to finish. Commit ratio also depends on several system conditions, including the number of active threads and the application nature. Commit ratio is lower when there are more threads because it is more likely to have collisions. Some applications are more sensitive to the number of active threads than others as the threads have more access to common data locations. We notice that both of the two factors are related to the number of threads. When there are more threads, there are more transaction attempts but with less success to commit (lower commit ratio).

We need an effective *adaptive concurrency control (ACC)* algorithm adjusting the number of concurrent threads based on observing the rate of transactional conflicts in order to maximize the throughput (commit rate) of the STM system. There are two challenges in designing such a control algorithm. First of all, it is non-trivial to determine whether a commit rate is regarded high or low. Second, it is tricky to obtain the live commit rate at runtime. It involves collecting total amount of commits, which forces the threads to share data across the processors.

## III. Dynamic Concurrency Control for STM on Multi-CMP Systems

In this section, we present our adaptive concurrency control protocol. The essence of the protocol is similar to flow control of communication which avoids overflowing the medium: we do not 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 transactional concurrency. We have neatly performed mechanism-policy separation. We first discuss the mechanism

TABLE II. Concurrency control variables

| variable | type | usage |
|---|---|---|
| quota | integer | concurrency quota—number of threads allowed to run transactions |
| active | integer | number of threads that have entered into transactions and not yet exited |
| peak | integer | the peak number of threads that runs transactions simultaneously |
| commits | integer | number of committed transactions |
| aborts | integer | number of aborted transactions |
| stalled | boolean | indicates whether some threads are stalled at transactions' entrance |

TABLE III. Concurrency Quota Mechanism

```
1   function onBegin
2   retry:
3       if $active \geq quota$ then
4           $stalled \leftarrow true$
5           yield
6           goto retry
7       end if
8       $active \leftarrow active + 1$
9       if $peak < active$ then
10          $peak \leftarrow active$
11      end if
12  end
13
14  function onCommit
15      $active \leftarrow active - 1$
16      $commits \leftarrow commits + 1$
17  end
18
19  function onAbort
20      $active \leftarrow active - 1$
21      $aborts \leftarrow aborts + 1$
22      yield
23  end
```

(Section III-A), and then the control policy named *Probe* (Section III-B).

### A. Concurrency Quota Control Mechanism

To facilitate concurrency control, we introduce several variables listed in Table II. These variables are shared among all the transactions in the system.

Table III shows the basic mechanism to operate on the *concurrency quota* concept. There is a daemon thread running in the background to control a system-wide quota parameter which limits the number of active threads to run on the STM. When a thread begins a new transaction (i.e. calling *stm_begin()* function), it has to check if the current number of active threads has reached the current *quota* allowed (line 3). If this is the case, it has to wait in place until either some active threads get exited transactions (either committed or aborted) or the current quota is lifted on demand by the background daemon. We apply an extra heuristic to the mechanism: we record the active thread count that once happens to be the highest into the variable *peak* (line 10). If the daemon finds that the current quota is even higher than this peak value, it will set *quota* to *peak*. This measure is to avoid *quota* being incremented beyond the highest concurrency requirement. Superfluous quota will simply oppose the virtue of putting a bound on concurrent thread count.

### B. The Probe Concurrency Control Policy

Table IV shows the algorithm of Probe. The name implies the algorithm is probing for a concurrency quota that optimizes *commits* by some "trial and error" method. In theory, as

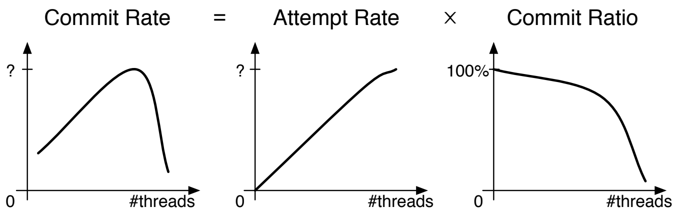

Fig. 2. Relation between commit rate, attempt rate, commit ratio and the number of concurrent threads

discussed in Section II-C, there exists an optimal active thread count that corresponds to the crest of the commit rate curve which looks bell-shaped as shown in Fig. 2. Note that in this policy, we use commit *rate*, i.e. the number of committed transactions per unit time, rather than commit *ratio* as other researchers like Ansari et al. [12] do. We quantify the unit time logically by a counter called *laps* which is being incremented per sampling event. To avoid reacting too fast, the policy proactively postpone quota updates until $commits + aborts \geq warmup$, where $warmup$ is a counter we add to record the number of daemon sleeping cycles that have passed until the system is "warmed up". A warmup phase is necessary to accumulate sufficient statistics upon which the ACC policy can be based in order to make correct decisions. Therefore, commit rate can be calculated by dividing *commits* by *laps*. We also keep record of the previous sample in variables *last_commits* and *last_laps*.

In reality, the optimal point for the current active thread count is floating—it may shift horizontally along the execution. 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 of commit rate (line 19), we know we are falling down the hill of commit rate and getting further away from the optimal. Therefore we should reverse the tuning direction from down to up, and keep incrementing the quota until we start to see another drop of commit rate, which is a signal for us to reverse the tuning direction from up to down.

### C. Metadata Zoning

We divide the compute cores into *zones* where cores in a *zone* belong to the same processor. As we have shown in Section II-A, data sharing is more efficient among cores in a zone, than cores from different zones. On each zone there is a separate set of concurrency control metadata. These metadata are not involved in conflict detection so this change does not affect the STM correctness. Each thread is set to belong to a zone and only access metadata related to the zone. This ensures the metadata remains exclusively shared among the cores in the same processor. The intra-processor data sharing allows the metadata to be read and updated more frequently, allowing higher amount of transactions to be processed within same amount of time. We also create an independent concurrency control daemons on each zone. Just in case transactions on different zones behave differently, the different daemons can react independently.

TABLE IV. The Probe concurrency control policy

```
1   set direction to down
2   **while** true **do**
3       sleep for a constant time (e.g. 5ms)
4       **if** $peak = 0$ and $active = 0$ **then**
5           **continue**
6       **else if** $commits + aborts < warmup$ **then**
7           $laps \leftarrow laps + 1$
8           **continue**
9       **else**
10          $laps \leftarrow laps + 1$
11      **end if**
12      **if** $peak < quota$ **then**
13          $quota \leftarrow peak + 1$
14          set direction to down
15      **else if** $quota = 1$ **then**
16          set direction to up
17      **else if** $commits/(commits + aborts) < 0.125$
18          set direction to down
19      **else if** $commits/laps$
                $< last\_commits/last\_laps$ **then**
20          set direction to reverse(direction)
21      **end if**
22      **if** direction is down **then**
23          $quota \leftarrow quota - 1$
24      **else**
25          $quota \leftarrow quota + 1$
26      **end if**
27      $last\_commits \leftarrow commits$
28      $last\_laps \leftarrow laps$
29      reset $peak$, $commits$, $aborts$, $laps$ to zero
30  **end do**
```

## IV. Affinity-Aware Thread Migration

In this section, we discuss a thread migration technique made to enhance the STM system for multi-CMP systems.

The aim of having thread migration among multiple processors is to better utilize the partially shared cache. The compute cores do not have access to all the cache units so that there are often some cache lines duplicated in different units, which reduce the effective cache size. There are also cache invalidation overheads when two threads on different processors modify the same data, stalling the thread for some considerable delays explained in Section II-A. By finding out the thread correlations, we can identify some threads having access to some common data. They can be rearranged to execute on cores of the same processor, reducing cross-die cache invalidation overheads and bringing about an overall speed improvement.

While runtime thread correlation is hard to track in a lightweight manner as our past experience tells [13], we can estimate the correlation by counting the conflicts between individual threads as two transactions conflict only when they access common data. Even if there can be a false conflict, they access some common pieces of STM metadata.

We compute and store the likelihood of threads to conflict in a 2D table of *pairwise contention intensity*. The original definition of *contention intensity*, proposed by Yoo and Lee [3], is a single weighted average value of a thread's conflict likelihood across time and ranges from $0$ (never conflict) to $1$ (always conflict). In our case, we borrow the concept for estimating inter-thread conflict probability. There are $n^2$ values for $n$ threads and each thread $T_i$ possess an array of $n$ numbers, $\{C_{ij} | 0 \leq j < n\}$, which represent how likely it is to be obstructed by thread $T_j$. While we could save half of storage by using a triangular matrix, we store $C_{ij}$ and $C_{ji}$ separately to eliminate inter-thread synchronization

TABLE V. AFFINITY-DRIVEN THREAD MIGRATION HEURISTIC

```
function onCommit($x$)
    for $j \leftarrow 0$ to $n$
        $C_{xj} \leftarrow C_{xj} \times \alpha$
    end for
end

function onConflict($x, y$)
    for $j \leftarrow 0$ to $n$ except $y$
        $C_{xj} \leftarrow C_{xj} \times \alpha$
    end for
    $C_{xy} \leftarrow C_{xy} \times \alpha + (1 - \alpha)$
end

function close($x, y$)
    return ($T_x$ and $T_y$ are on same processor)
end

function distant($x, y$)
    return ($T_x$ and $T_y$ are on different processor)
end

function pickPair
    for $i \leftarrow 0$ to $n$
        $D_i \leftarrow \sum\{C_{ij} + C_{ji} | distant(i, j)\}$
              $- \sum\{C_{ij} + C_{ji} | close(i, j)\}$
    end for
    $benefit \leftarrow$ max $\{D_i + D_j - 2(C_{ij} + C_{ji}) | distant(i, j)\}$
    return $(i, j, benefit)$
end

thread migrationDaemon
    while true do
        $(i, j, benefit) \leftarrow pickPair$
        if $benefit > (1 - \alpha)$ then
            Swap $T_i$ and $T_j$
        end
        sleep for a constant time (e.g. 50ms)
    end
end
```

and let each thread have exclusive write access to its array. The 2D array can be taken as an directed adjacency matrix which represents the edge presences and weights. Our aim is to reduce the sum of edges that span across different CMPs. Upon successful partitioning, we have fewer closely related threads running on different CMPs. Graph partitioning problem is NP-complete and less expensive heuristics are required. We derive a solution based on the work of Kernighan and Lin [14]. The heuristic accepts a graph partitioned in two sets, $A$ and $B$; in each of the iterations, a pair of vertices from two partitions is picked and swapped. In our situation, in order to avoid stressing the inter-processor communication channel, having fewer thread migrations at each step is preferable. Therefore, only a single pair of thread is taken within a time slice. This further simplifies our final heuristic as shown in Table V.

When a transaction commits successfully, it reduces its pairwise conflict intensities against all the threads with the function *onCommit*.[4] The values are decreased by a defined factor $\alpha$, which we set to $0.9$ in this paper. In contrast, when a transaction $T_x$ is obstructed by another transaction $T_y$, the function *onConflict* is called so that the contention intensity of the pair $(x, y)$ is increased. Finally, a daemon thread calls *pickPair* periodically (per 50ms, say) to select a pair of threads to swap over. The array $D$ specifies how much benefit (reduction of cross-partition edges) is obtained by migrating a single thread to another processor. The overall benefit from swapping two threads $T_i$ and $T_j$ is $(D_i + D_j) - 2(C_{ij} + C_{ji})$,

[4]To facilitate fast commit in the implementation, we adopt a lazy update mechanism which updates one variable only.

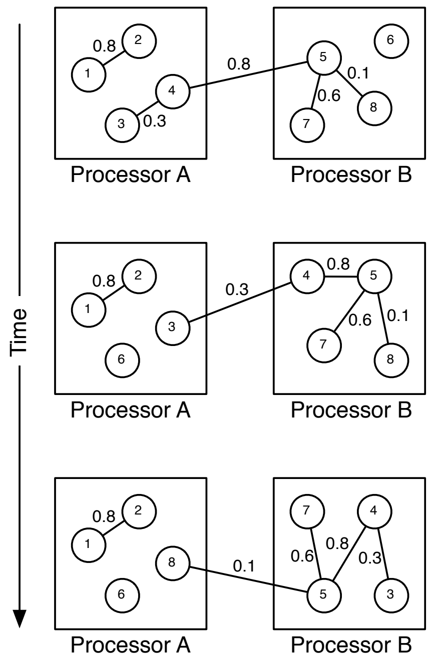

Fig. 3. A thread migration example

where the latter of the expression is due to the fact that we do not cancel the inter-partition edges between the two swapped threads.

A thread migration example is shown on Fig. 3. The integers inside the nodes represent the thread identifiers, and the edges are marked with contention intensity values of the individual thread pairs. In the first transition, thread $T_4$ is picked to migrate to processor B. We prefer migrating $T_4$ instead of $T_5$ because the latter is much more bonded with its neighbors ($D_5$ is negative). We pick $T_6$ to replace $T_4$'s position because it is least bonded with its neighbors. In the second transition, $T_3$ is picked because $D_3$ is the biggest among the threads on processor A. $T_8$ is picked to replace the vacancy. Although we now schedule $T_8$ and its neighbor $T_5$ on different processors, we observe the overall cross-boundary edge value is substantially reduced from $0.8$ to $0.1$.

Although the *pickPair* process is having O($n^2$) time complexity, we do not regard it as a serious burden because the procedure is only executed once at a time and it does not delay or pause execution of most threads until it makes a decision. The heuristic only sets processor affinities of particular threads and it does not interfere with the conflict detection mechanism, so the correctness of execution is not affected.

The migration policy is currently designed for STMs running on dual-processor systems, which are commonly used as workstations and general-purpose servers. When there are more than two processors, we can run *pickPair* in multiple iterations to swap threads around different processors, or apply multilevel partitioning. However, their implementation is out of the scope of this paper.

## V. IMPLEMENTATION

We implement all the above-mentioned techniques in TinySTM 0.9.5 [10], one of the state-of-the-art word-based

STMs. We pick this specific version in order to have a fair comparison with related work on concurrency control policies that have gone open-source and bundled in this version.

Thread stalling due to insufficient concurrency quota is realized by spins over `sched_yield()` system call. We prefer this system call rather than spin waiting as it gives the operating system authority to preempt a thread for others. This is especially useful when there are more threads than number of hardware thread units [4]. Daemons wake up periodically by `usleep()` system call. A daemon thread does not have enough workload to keep a core fully utilized. Once it finishes its decision making, it sleeps voluntarily, giving way to other computational threads. While sleeping is not absolutely accurate in timing, we find it accurate enough for the daemons to handle with time-related statistics properly. We therefore avoid calling other system calls such as `gettimeofday()` for timing purpose. Zone switching is implemented with Linux-specific system call `sched_setaffinity()`. When the system executes without the migration policy, each thread make this system call once only when it spawns in order to set it to a zone. When there is a migration policy, a thread make this system call every time it observes a signal (updated shared variable) to switch zone.

In order to trace the conflicts between the threads, we reserve some bits in the timestamp entries. When a transaction unlocks a timestamp, it also leave its thread number as part of the timestamp. As a timestamp entry is a long machine word (64 bits on modern computers), reduction of a few bits does not cause clock rollover problems. TinySTM also comes with mechanism to handle the rollover problem, if it ever happens. The correctness and consistency of metadata variables *active*, *quota* and *peak* are crucial to keep the system from deadlock. For example, when *quota* is set to be 1 while *active* is miscomputed to be 1 as well, the system enters a state that threads cannot start new transactions. The variables are therefore stored as bits in a 64-bit integer and modified through compare-and-swap (`cmpxchg8b`) instructions. Upon update of any of these variables, the thread also ensures previous read values are still valid. The variables *stalled*, *commits* and *aborts* are not protected by similar measures. Slight errors on these variables will not affect decision making of the policies. We trade the absolute accuracy for faster commit procedures.

Although we are adding an adaptive concurrency controller on top of a blocking STM, our part of implementation itself is guaranteed obstruction-free [6] as our empirical studies reveal that any blocking concurrency control (i.e., using some locks to protect metadata) can seriously degrade the STM performance.

## VI. Performance Evaluation

In this section we evaluate our implementation and compare it with two third-party concurrency control policies by Yoo and Lee [3] and Dragojević, et al [1], which are referred to as *Yoo* and *Shrink* respectively in later context.

### A. Evaluation Platform and Methodology

Our experiments were conducted on a Dell PowerEdge M610 blade server equipped with two Intel Xeon E5540 Quad-core 2.53 GHz processors and 36 GB of DDR3-1066 ECC memory. Both the Turbo-Boost and Hyper-Threading options were enabled. With a total of eight cores, and two threads per core, the server can support 16 simultaneous threads. We pick Fedora Release 11 as the operating system for evaluation.

We evaluate our solutions using the STAMP benchmark suite [15] with modifications tailored for TinySTM. For comparison purpose, we obtained implementation of Yoo and Shrink, tied with TinySTM 0.9.5, from the website of *Distributed Programming Laboratory of EPFL* [16].

We run the ten test cases unmodified to see how much speedup the four concurrency control policies, namely Probe-Z, Probe-ZM, Shrink and Yoo, can gain over the plain execution without concurrency control. The first two policies (with a suffix Z) are our zone-based versions of the Probe policy, i.e. they have a separate set of metadata and control daemon on each processor. Probe-ZM implements also the thread migration policy besides zoning. For comparison purpose, we also include the original Probe, which has only one system-wide control daemon and no partitioning of threads into zones. For ease of reference, we use the term ACC (adaptive concurrency control) in later context to collectively refer to any of these policies.

While the system supports only 16 simultaneous threads, we intentionally have a case of running 32 threads to see how the adaptive concurrency control protocols handle the adverse condition of excessive threading. Some applications show high discrepancies in execution time across runs. We handled this by running each of the tests five times and taking the average.

### B. Experimental Results

Fig. 4 shows speedup charts of different TM benchmarks in the STAMP suite, with and without ACC. Each separate chart corresponds to one application. (There are two test cases for each of *kmeans* and *vacation*: low (1) and high (2) in terms of inherent contention). The $x$-axis represents the static initial thread count (ITC) spawned by the benchmark. Except "original" (i.e., the baseline execution with ACC disabled), the actual number of active threads is under the influence of the ACC and changes from time to time.

TinySTM generally shows increasing speedup when ITC is between 2 to 8. Speedup drops when ITC reaches 16 or 32, depending on application nature. STAMP is a transactional benchmark suite with applications of different natures. There are several relatively non-scalable benchmarks in the STAMP suite— *kmeans*, *ssca* and *yada*, as well as some scalable benchmarks such as *genome*, *labyrinth* and *vacation*. We observe irregular speedup graphs in *bayes*.

With the ACCs enabled, the speedup drops less rapidly when there are excessive threads running. However, there are also cases where ACCs result in degraded STM performance for a number of benchmarks, when there are moderate amount of initial threads.

Table VI shows the best-case, worst-case and average gain in speedup of various ACCs, compared to the baseline execution without ACCs. We have skipped *bayes* from comparison because its execution time depends on the order of graph nodes being processed, instead of true performance of the STM framework [15]. It is encouraging to see that Probe-Z and

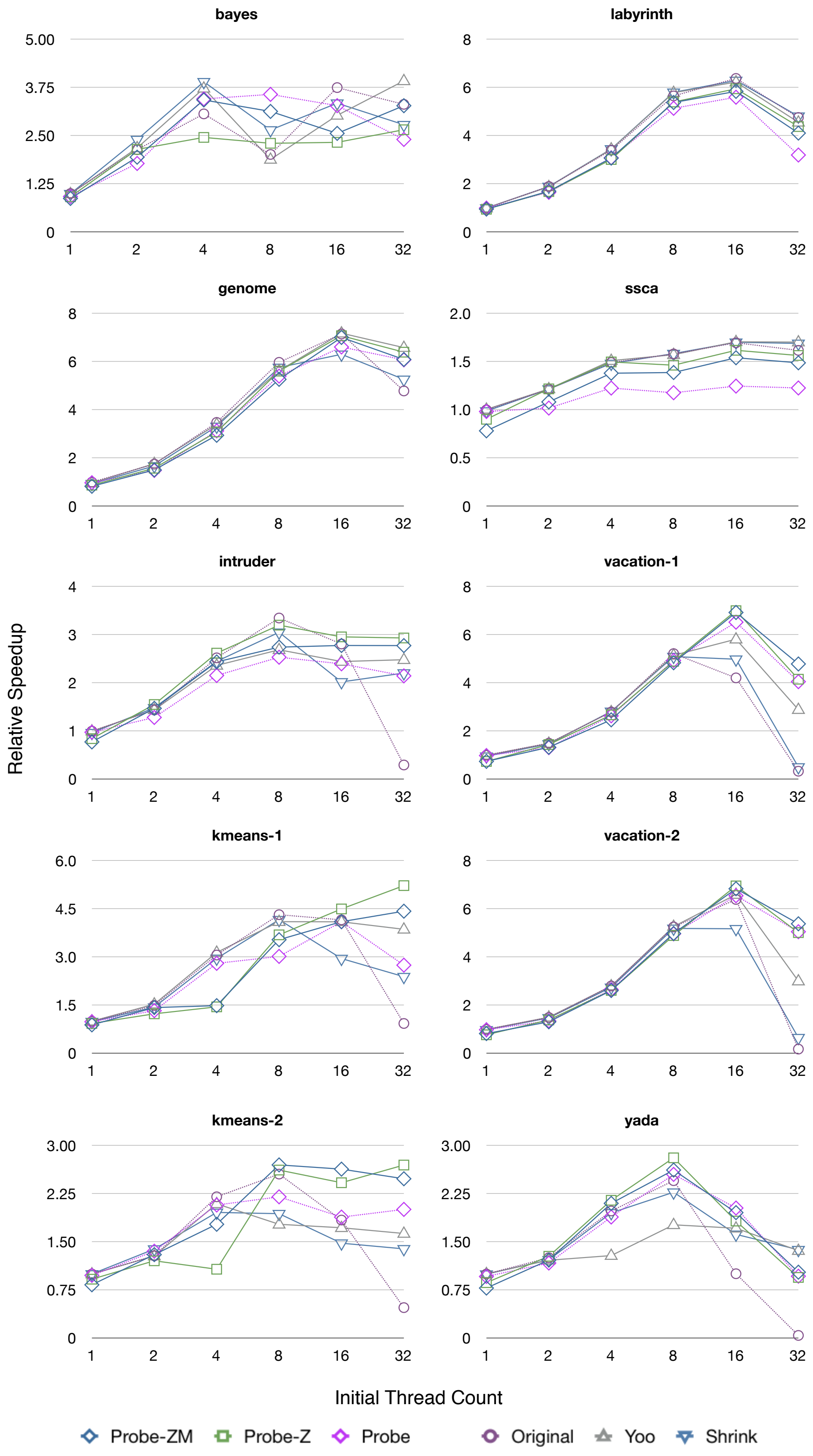

Fig. 4. Speedup of STAMP applications under different concurrency control policies

TABLE VI. EFFECTIVENESS OF VARIOUS ACC TECHNIQUES ON STAMP BENCHMARKS

| Policy | Best-Case | Worst-Case | Average |
|---|---|---|---|
| Probe | +101.79% (yada) | -26.51% (ssca) | +11.03% |
| Probe-Z | +81.59% (yada) | -6.73% (labyrinth) | +21.04% |
| Probe-ZM | +94.70% (yada) | -9.18% (ssca) | +20.08% |
| Yoo | +70.50% (yada) | -12.95% (intruder) | +9.95% |
| Shrink | +61.02% (yada) | -28.96% (kmeans-1) | -3.23% |

Probe-ZM outperform other ACCs in most of the test cases, and obtain 21% and 20% average speedup respectively.

## C. Discussion on the Results

Our policies generally perform the best on *vacation*, which is a travel agency simulator. Clients make random reservations so that the data accesses in different transactions are not necessarily related. Both our policy and Yoo's algorithm detect undesirable commit rate and ratio, and lower the concurrency accordingly. While we believe Shrink detects undesirable commit ratio as well, the nature of random data access makes the hotspot detection useless. Nevertheless, the hotspot detection hinders the loser threads' progress, and hence slightly reduces the contention.

Our policies bring about a slight slowdown to *ssca* while Yoo and Shrink have performance on a par with the baseline. This is due to the already saturated communication channel for cache-level data transfer. Any additional implementations that share data cross cores can really slow down the system significantly. Due to the high commit ratio, both Yoo's and Shrink policies do not react to reduce concurrency.

Our Probe policy, as well as Shrink, achieve excellent improvement for *yada* when there are 16 initial threads. These algorithms do not rely purely on commit ratio for decision making. As *yada* has a commit ratio well below 20%, other ACCs are confused and get the program over-serialized.

Performance of Probe and Probe-Z is very similar for a lot of applications. However, Probe-Z is much more well-rounded, providing much better worst-case performance. Among the STAMP applications, *intruder*, *ssca*, and *kmeans* have by nature high commit rates. Probe cannot perform well for these applications because the concurrency control metadata have to be frequently transferred between the two disjoint processor caches. By partitioning the threads into zones, the performance degradation is greatly relieved.

To further investigate the difference in performance of the ACCs, we carried out an analysis on some of the benchmark applications, and the results are shown in Fig. 5. The performance of the *yada* application gets quite close to the sweet spot with 8 threads although the commit ratio is well below 10%. Our Probe policy improves the situation by stalling a few threads, getting the program accelerated by about 10%. Shrink also performs well by stalling some threads. Before stalling a thread, it performs hotspot detection to ensure only transactions accessing hotspots are stalled. Since it finds only some amount of hotspots, the system stalls only a few threads and achieves a good result. Yoo's policy does not work well with *yada*. It keeps watching the contention intensities of individual threads and is thus mislead by the commit ratios to stall too many transactional threads. While they achieved much higher commit ratios than Probe and Shrink, they have actually dragged down the commit rate, and hence lengthened the program execution.

For *kmeans* executed with 16 threads, there exists severe contention so that the commit ratio is below 25% when ACC is not activated. This application performs clustering on multi-dimensional data points. In each transaction, the center of a cluster is modified. As there are only a few clusters, they become memory hotspots in the program. By adaptively stalling different amounts of threads, Probe drastically improves the commit rate. Yoo keeps the commit ratio constant, at about 65%, which is the same as that in *yada*. Shrink keeps the commit ratio at about 80%, which is close to the threshold of activating hotspot detection. As all the transactions make access to hotspots, a transaction is definitely stalled when the thread-local contention drops below the threshold.

The performance results of Probe-Z and Probe-ZM on STAMP benchmarks are quite similar. This is because there are not any specific thread correlations in the STAMP benchmark inputs. It would be worse if the system spends extra effort on detection and migration but ends up achieving nothing better in terms of cache sharing. To demonstrate the potential of the system, we develop a micro-benchmark called *dual red-black tree*. In this program, each thread is given either a group of odd numbers or a group of even numbers, which are to be inserted into two red-black trees. Without proper thread placement, the threads will contend for the same memory location to make their updates. As shown in Fig. 6, Probe-ZM reduces cross-partition contention intensity and makes the system scale almost linearly when there are 8 threads. In comparison, the original TinySTM yields less than a speedup of 4 times in the same situation.

## VII. RELATED WORK

Ansari et al [2], [12] propose a P-only control algorithm for tuning the active thread count according to the observed commit ratio. They presented preliminary benchmarking results, tested with eight threads only, to show the effectiveness of their algorithm compared with other simple adaptive concurrency control schemes. In contrast, we propose a new policy (Probe) that observes commit rate instead of commit ratio, and also conduct experimental evaluation of a bigger scale and a wider range of benchmarks.

Yoo and Lee [3] proposed another mechanism to adjust number of active threads. Unlike the work by Ansari, Yoo's policy does not require heuristic data sharing among threads. It computes *contention intensity* on each thread, which, when above a threshold, the thread acquires a common lock before starting a new transaction. Meanwhile, counting number of conflicts is not reliable as large amount of conflicts (e.g. in Yada) may mislead the policy to unnecessarily serialize the transactions. Our Probe policy disregards the rollback counts and does not suffer from the same problem.

Shrink [1] assumes conflicts are induced by memory hotspots. It adaptively activates hotspot detection when there are repeated conflicts encountered by a thread. Transactions that are known making accesses 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 problem is purely random, all these efforts do nothing helpful

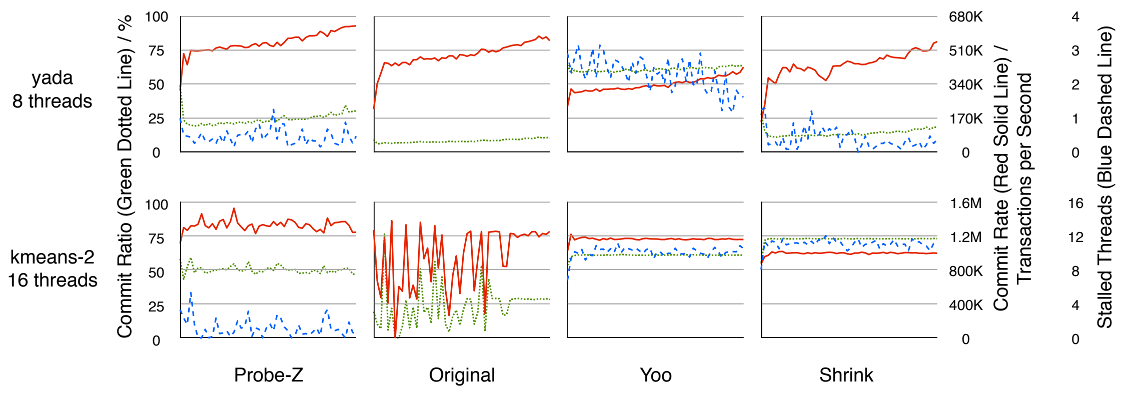

Fig. 5. Commit ratio, commit rate and number of stalled threads of some TM applications

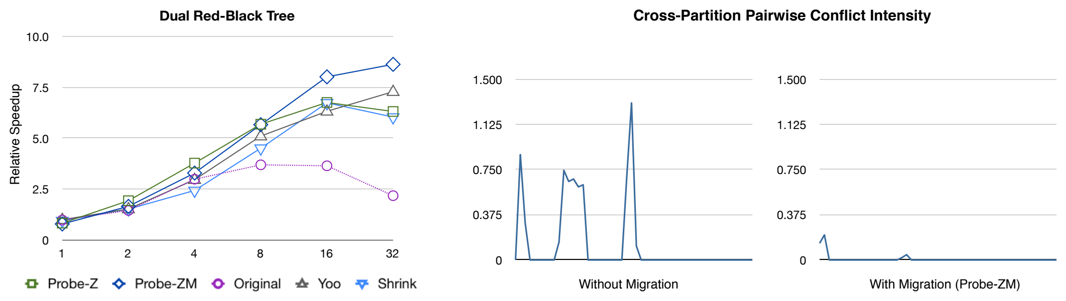

Fig. 6. Dual red-black tree microbenchmark scalability and cross-partition contention intensity

but waste even more computing time, as shown in Vacation benchmark. On the contrary our heuristics handle this case properly by reducing number of active threads globally.

Key-based adaptive transactional memory executor [17] reduces conflicts by introducing the concept of *key*. Transactions modifying similar data are given similar keys. The executor schedules unrelated transactions (of dissimilar keys) on different processors for maximal parallelism and related transactions on the same processor for conflict prevention plus better cache utilization. Unfortunately, the design lacks a well-defined strategy to assign proper keys to transactions.

Dependence-aware transactional memory (DATM) [18] is a recently proposed model for increasing concurrency of memory transactions without complicating their interface. DATM manages dependencies between conflicting, uncommitted transactions so that they commit safely. DATM accepts all conflict-serializable concurrent interleavings. DATM outperforms TL2 by 4.86 times at 16 threads for Vacation. However it lacks evidence for other applications, especially the ones with low commit ratio such as Kmeans. The model also has to deal with deadlocks that can arise in commit protocol for various reasons.

Maldonado, et al [4] proposes various methods to schedule transactional threads. They extensively evaluated various methods of scheduling, including spin-locks, condition variables (CVs), and extending the kernel schedulers. For instance, they noticed using CVs to wake up loser threads adds huge overhead to the commit procedure. While transaction-aware schedulers perform better than user-space contention management in excessive threading situations, most of the implementations perform similarly in situations where there are more threads than cores.

## VIII. Conclusion and Future Work

In this paper, we demonstrate an advanced software transactional memory (STM) tailored to platforms of multiple chip multiprocessors (CMPs). To optimize the throughput of the STM, a cache-aware design of dynamic concurrency control is important for regulating the rate of spawning transactions with low overheads on a multi-CMP system. The concurrency control aims for seeking the sweet spot on commit rate by varying the number of active threads, and is found more robust than other solutions which might get misled by low commit ratios. We found that control overheads are very sensitive to the underlying cache sharing and illustrate the zoning technique to localize the traffic of metadata sharing across cores. We also refine our implementation with a novel thread migration policy that considers pairwise contention intensities and swaps threads in a sense to improve the cache affinity. In future, we may consider new adaptive scheduling policies that observe more STM parameters for smarter decisions, including transaction length, read-write ratio, etc.

## Acknowledgment

This research is supported by Hong Kong RGC Grant HKU 718011E and China 863 grant 2014AA01A302.

## References

[1] A. Dragojević, A. Guerraoui, R. amd Singh, and V. Singh, "Preventing versus curing: Avoiding conflicts in transactional memories," in *Pro-*

*ceedings of the 28th ACM Symposium on Principles of Distributed Computing*, 2009, pp. 7–16.

[2] M. Ansari, C. Kotselidis, K. Jarvis, M. Lujan, and I. Watson, "Advanced concurrency control for transactional memory using transaction commit rate," in *Proceedings of the 32rd International Conference on Distributed Computing Systems*, 2008, pp. 522–529.

[3] R. Yoo and H. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, 2009, pp. 169–178.

[4] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. Lawall, and G. Muller, "Scheduling support for transactional memory contention management," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010, pp. 79–90.

[5] K. Chan, K. T. Lam, and C. L. Wang, "Adaptive thread scheduling techniques for improving scalability of software transactional memory," in *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing Networks*, 2011.

[6] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction free synchronization: Double ended queues as an example," in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003, pp. 522–529.

[7] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott, "Lowering the overhead of nonblocking software transactional memory," in *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[8] R. Ennals, "Software transactional memory should not be obstruction-free," Intel Research Cambridge, Tech. Rep. IRC-TR-06-052, 2005.

[9] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *Proceedings of the 20th International Symposium on Distributed Computing*, 2006, pp. 194–208.

[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*, 2008, pp. 237–246.

[11] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Programming Language Design and Implementation*, 2009, pp. 155–165.

[12] M. Ansari, C. Kotselidis, K. Jarvis, M. Lujan, C. Kirkham, and I. Watson, "Adaptive concurrency control for transactional memory," in *Proceedings of the 1st workshop on Programmability Issues for Multi-Core Computers*, 2008.

[13] K. T. Lam, Y. Luo, and C.-L. Wang, "Adaptive sampling-based profiling techniques for optimizing the distributed JVM runtime," in *Proceedings of the 24th IEEE International Symposium on Parallel Distributed Processing*, 2010.

[14] B. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[15] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Standford transactional applciations for multi-processing," in *Proceedings of the 2008 IEEE International Symposium on Workload Characterization*, 2008, pp. 35–46.

[16] LPD-EPFL, "How good is a transactional memory implementation," http://lpd.epfl.ch/site/research/tmeval, accessed on 30 July 2010.

[17] T. Bai, X. Shen, C. Zhang, W. Scherer III, C. Ding, and M. Scott, "A key-based adaptive transactional memory executor," in *Proceedings of the 2007 IEEE International Symposiuum on Parallel and Distributed Processing*, 2007, pp. 1–8.

[18] H. Ramadan, C. Rossbach, and E. Witchel, "Dependance-aware transactional memory for increased concurrency," in *Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 246–257.