Profiling Transactional Applications

Vincent Gramoli [1] Rachid Guerraoui [2] Anne-Marie Kermarrec [3]

[1] NICTA and University of Sydney, Australia
vincent.gramoli@sydney.edu.au
[2] EPFL, Swtizerland
rachid.guerraoui@epfl.ch
[3] INRIA, France
anne-marie.kermarrec@inria.fr

Abstract. What does it mean for two transactional applications to be similar? We address this question in this paper by highlighting four distinctive features of transactional applications: (1) the transaction size, i.e., the average number of memory accesses of the transactions; (2) the read-write ratio, i.e., the ratio between the number of accesses that modify the data and those that do not; (3) the contention, i.e., the number of concurrent accesses to the same shared data, such that at least one of these accesses is a write; (4) the uniformity, i.e., the extend to which transactions access distinct objects. We show that the similarity between an application A and an application A' can be derived from these features and can be used to determine which concurrency control implementation works best for A based on having tested which worked best for A'. We convey the accuracy of the profiling and predictions based on a study with six workloads and ten concurrency control mechanisms.

Keywords: Recommendation system, performance, collaborative filtering, concurrency control

1 Introduction

Profiling applications a priori is key to their effective deployment. The idea is very simple: given some characteristics of an application computed a priori, one can choose the best deployment scheme for the application without having to go through exhaustive testing schemes that might sometimes not even be possible. Profiling is particularly appealing for concurrent applications. In this case, deployment includes, among other things, the choice of the underlying concurrency control mechanism to ensure consistency despite concurrent accesses to shared data. Adopting the wrong mechanism can hamper scalability and impact performance by several orders of magnitude [13]. In addition, there are multiple ways of combining existing mechanisms and going through exhaustive testing may simply be impossible. Hence the need for profiling.

Yet, profiling concurrent applications is very challenging. This difficulty stems from the inherent nondeterminism of concurrent applications as well as from the layout of the architecture where these programs run [8]. Furthermore, manufactured hardware evolves rapidly by, for example, adopting different cache coherence protocols [20] or multiplying the number of cores.

In this paper, we focus on transactional applications. We highlight four distinctive features: (1) the transaction size, i.e., the average number of memory accesses of the transactions; (2) the read-write ratio, i.e., the ratio between the number of accesses that modify the data and those that do not; (3) the contention, i.e., the number of concurrent accesses to the same shared data, such that at least one of these accesses is a write; and (4) the uniformity, i.e., the extent to which transactions access distinct objects. We show that these features can be computed for an application A and are sufficient to determine its distance from an application A'. In turn, this distance can help predict which concurrency control mechanisms would best fit A based on results obtained on A'.

In some sense, this is like highlighting distinctive features of a person P that would help compute similarities with another person P' and help predict which movies P would like most based on those that P' enjoyed most. In our context, we show for instance that if a new application is similar to others, then the concurrency control mechanism that is known to benefit these latter applications could intuitively benefit the new one as well. Or we can filter out inappropriate concurrency controls based on the observed similarities between applications and their individual performance.

This collaborative filtering technique was initially used to compute the similarities between documents [26] and was more recently applied to measure similarities in large data sets [27]. The key idea is to filter information based on the collaboration of multiple participants or data sources. This technique is popular for its effectiveness in recommendation systems: by collecting tastes and preferences of many users regarding multiple items, the system can recommend an item that one user is likely to prefer. Although similar, the problem we tackle is not to recommend items based on their similarities but rather to suggest concurrency control to applications based on application similarities. The benefit of our approach is that the profile of an application is sufficient to select its most suitable concurrency control, simply by comparing this profile against profiles of existing applications that were previously tested. In particular, it is not necessary to test the new application to identify the concurrency control.

Using Synchrobench [13], we show experimentally that our approach is beneficial to even a small set of applications by precisely identifying the discriminating criteria. In particular, we show that the size of operations, the ratio of shared write accesses over shared read accesses, the contention and the uniformity of memory regions these applications access are effective criteria to compute similarities between applications and to suggest concurrency controls that boost performance. We evaluate our solution on 6 benchmarks for which workloads exhibit a wide range of behaviors with respect to these criteria.

In addition, we applied our approach to 10 concurrency control mechanisms. These mechanisms include various transactional algorithmic designs that are known to affect performance [11]: concurrency control mechanisms that acquire locks eagerly (encounter-time locking), lazily (commit-time locking), that use invisible writes, or visible (in-place) writes. These mechanisms are further refined using different policies similar to existing contention managers [17,25,24,16]. These policies include "kill-attacker" that always aborts the transaction that causes the conflict, the exponential backoff strategy that forces every restarting transaction to wait a period that increases exponentially with its number of restarts and a delaying contention manager that consists in restarting a transaction, which aborted due to an unsuccessful attempt to acquire a lock, only after the lock has been released.

Our experimental evaluation compares the performance in terms of throughput and abort rates of all benchmarks and compute their distance using the *cosine similarity* [26] of workloads based on the aforementioned criteria. Our conclusion is fourfold. First, our results confirm that for a given benchmark and depending on the concurrency control mechanism used in the benchmark, the performance significantly varies for the same update ratios, hence motivating the need for our solution. Second, our results indicate for instance that for data structures that share similar criteria, like red-black trees and skip lists, the same concurrency controls can benefit or penalize both corresponding benchmarks. Third, for benchmarks that have notably different profiles a concurrency control mechanism benefiting one may be substantially detrimental to the other. Finally, seemingly identical benchmarks may have different profiles due to the way they were tuned, performing differently with the same concurrency control.

Section 2 introduces the criteria to draw the profile of each concurrent workload, hence allowing us to compute similarities between them. Section 3 describes the transaction algorithms and contention management mechanisms resulting in the 10 concurrency control mechanisms we present. Section 4 depicts the performance results of our benchmarks as the throughput and abort rates when using each of the proposed mechanisms. Section 5 discusses how to extend our solution to implement a fully automated framework that refines application similarities based on previous runs of the applications. Section 6 presents the related work and finally Section 7 concludes the paper.

2 Workload Profiles

We define the *profile* of each workload as a set of four values, each representing its characteristic according to a distinct attribute or *dimension*.

- **Transaction size:** the transaction size captures the number of memory accesses executed as part of the same transaction between its last (re-)start and its commit.
- **Write/read ratio:** the mean ratio of the number of transactional write accesses over the number of transactional read accesses executed by a single transaction between its last (re-)start and its commit.

- **Contention:** the chance of conflicts inherent to this workload. Note that the contention is not related to the write/read ratio as two transactions executing mostly writes on disjoint data may not contend.
- **Uniformity:** the level of uniformity in the distribution of transactional accesses over memory locations. A lower value indicates skewness so that the same few locations are more likely accessed by any transaction than other locations. Note that the same workload can be skewed and not contended.

As we explain below, these four dimensions allow us to compare statically different workloads based on the distance between the vectors of their profile.

2.1 Workloads

To compare concurrency control, we evaluate six workloads freely available with Synchrobench [13], a micro-benchmark suite for synchronization techniques. The first workload is a hash table that maps a key to a value in constant size buckets implemented as linked lists and where n threads execute the three operations similarly to the list-based set workload. It features simple transactional operations `put`, `delete`, `contains`, that consists of adding, removing an element from a set and checking for the presence of an element in the set, respectively. The workload consists of spawning n threads that repeatedly execute randomly these transactional operations with a proportion of `put` and `delete` over `contains` specified with an update ratio u. These operations execute on a hash table initialized with a given number of elements (indicated by parameter i), each operation takes a value uniformly at random in a range of r possible values. Other workloads include a linked list, a red-black (RB) tree, an AVL speculation-friendly (AVL SF) tree [5] and a skip list where n threads execute operations with the same distribution. The last workload is a double-ended queue that consists of an array where values are always `enqueue`d at the head and `dequeue`d from the tail, hence implementing a queue abstraction (the update ratio is thus always 100%).

2.2 Profile-Based Comparisons

To identify the profile of each of these applications, we ran experiments using Synchrobench [13] and observed the size of transactions, compared the size of the *read-set* or the number of shared read accesses within the same committing transaction to the size of the *write-set* of the number of shared write accesses per committing transactions. The profile of each workload namely the hash table, the list-based set, the red-black tree, the speculation-friendly tree, the skip list and the double-ended queue are depicted in Table 1. Given this profile, we can measure the distance between two workloads depending on the offset between their coordinates. We deduce the profile of each workload by observing the length of transactions, the proportion of write vs. read accesses to the shared memory, the frequencies at which two transactions access same shared locations and the distribution of accessed locations.

Workload	**Hash table**	**Linked list**	**RB tree**	**AVL SF tree**	**Skip list**	**Deque**
Tx size	-	++	+	+	+	-
Write/read	+	-	-	-	-	++
Contention	-	-	+	-	+	++
Uniformity	++	+	-	-	+	-

Table 1. Profile of the workloads depending on the transaction size, the proportion of write over read and the amount of conflicts

Linked list transactions are larger than others as the number of accesses per transaction is linear in the number of elements and the list contains as many elements as other structures. Empirically, we observed for 2^{16}-sized data structures that the average linked list transaction size was $66K$. We also confirmed that transactions were very small on constant access time data structures: we observed 5 shared accesses per transactions on hash tables and queues. Finally, we found slightly varying transaction size on logarithmic access time data structures: 61 for the skip list, 40 for the red-black tree and 23 for the speculation-friendly tree. These differences are reported in the row 'Tx size' of Table 1.

No operation in any workload has a number of write accesses linear in the number of elements, this number is either constant in the linked list, queue, hash table sizes and is typically logarithmic in the skip list size and it may be logarithmic in the tree size when restructuring is involved. Experimentally, we observed that the linked list has a ratio of write over read shared accesses of $\frac{1}{32K}$ whereas the hash table experience a ratio that varies from $\frac{1}{6}$ with a load factor of 10 to $\frac{4}{5}$ with a load factor of 1. The deque ratio is $\frac{2}{3}$, the skip list ratio is $\frac{1}{30}$, the AVL SF tree ratio is $\frac{1}{20}$ and the RB tree ratio is $\frac{1}{7}$.

The contention is induced by the probability for two operations to access the same shared location so that at least one of these accesses is a write. This is very likely for the deque as all operations write to the same 3 (on average) locations located either at the head or the tail of the queue (depending whether they enqueue or deque). Finally, the uniformity indicates the skewness in the distribution of access locations among all locations. The deque as well as trees are highly skewed as their head, tail and root are the most accessed locations. The hash table experiences the highest uniformity because the hash function tends to balance the accesses among all buckets.

Non-discriminating criteria. If we represent the six workloads as three-dimensional vectors by ignoring the fourth criterion (uniformity) $(x, y, z) = (1,2,1),(3,1,1),(2,1,2),(2,1,1),(2,1,2),(1,3,3)$ in the column order in which they are listed in Table 1 then the Euclidean distance

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

is 0 between skip list and red-black tree, the distance between each of these two and the AVL SF tree is 1. The linked list is closer to the AVL SF tree (with distance 4) than the skip list or red-black tree (5), the hash table is at distance

3 from the skip list or red-black tree, and at distance 2 from the AVL SF tree and the two furthest workloads are AVL SF tree and deque with a distance of 9.

This selection of three criteria does not help differentiating the skip list and the red-black tree. Note that this observation holds regardless of the distance metric we choose because the skip list and the red-black tree would share exactly the same profile. To refine our profiles and differentiate these two workloads we have to take into account the fourth criterion, namely uniformity.

Discreminating criteria. Picking only three criteria may not be discriminating enough. If we refine our profiles using the four criteria, we can represent the six workloads with four-dimensional vectors

$$\begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} \text{ as } \begin{pmatrix} -1 \\ 0 \\ -1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \\ -1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \\ 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \\ -1 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \\ 0 \\ 0 \end{pmatrix} \text{ and } \begin{pmatrix} -1 \\ 1 \\ 1 \\ -1 \end{pmatrix}.$$

Cosine similarity. Given the fourth criteria, we can now refine our notion of distance by taking the cosine similarity to compare the direction of these vectors. Note that the cosine similarity is more effective to distinguish between profiles that do not share a majority of coordinates than existing alternatives like the Euclidean distance, the Pearson Correlation Coefficient or the Tanimoto Coefficient [27].

The cosine similarity between two vectors $v_1 = (w_1, x_1, y_1, z_1)$ and $v_2 = (w_2, x_2, y_2, z_2)$ is:

$$\frac{v_1 \cdot v_2}{||v_1|| \times ||v_2||} = \frac{w_1 \times w_2 + x_1 \times x_2 + y_1 \times y_2 + z_1 \times z_2}{\sqrt{w_1^2 + x_1^2 + y_1^2 + z_1^2} \times \sqrt{w_2^2 + x_2^2 + y_2^2 + z_2^2}}.$$

The cosine similarities between each pair of profiles is depicted in the symmetric Table 2.

	Hash table	**Linked list**	**RB tree**	**AVL SF tree**	**Skip list**	**Deque**
Hash table	1	0	-0.41	0	0	-0.29
Linked list	0	1	0.4	0.67	0.58	-0.87
RB tree	-0.41	0.4	1	0.81	0.7	0
AVL SF tree	0	0.67	0.81	1	0.58	-0.5
Skip list	0	0.58	0.7	0.58	1	-0.5
Deque	-0.29	-0.87	0	-0.5	-0.5	1

Table 2. Similarities between workloads in term of the cosine similarity of their profile

3 Algorithms for Concurrency Control

To identify whether a specific concurrency control algorithm can benefit a particular workload, we choose four different transactional memory (TM) algorithms and four different contention managers algorithms (CM). The three TM algorithms are the following:

- **EagerAcq**: eager acquirement is a technique that consists of acquiring a lock on some shared variable immediately. To allow for read sharing, our transactions simply acquire a lock on the variables they attempt to update. With eager acquirement, the transaction acquires the lock before deciding to commit or abort.
- **InvWrite**: invisible write is the technique of deferring the write to the commit time of the transaction. With this approach a transaction that writes some shared variables lets concurrent transactions access the same variables between the time it "speculatively" writes them and the time it commits. This is the technique used in TL2 [9], it shortens the average protection duration of transactions by postponing all lock-acquirements to the commit phase.
- **WriteInPlace**: as opposed to invisible writes, writing in place consists of effectively updating the memory before reaching the commit. If the transaction aborts, then some compensating actions must be executed to roll-back the unsuccessful updates.

We also choose three different contention manager (CM) algorithms:

- **KillAttacker**: this strategy consists simply of choosing to abort the transaction that detects the conflict (the attacker) rather than the other conflicting transaction (the victim). A transaction is restarted immediately after it aborts.
- **ExpBackoff**: this strategy forces an aborting transaction to wait a duration that increases exponentially with the number of aborts before restarting. Once a transaction at some process commits, the next transaction executed by the same process starts without any delay.
- **Delay**: this strategy uses the same KillAttacker strategy without restarting a transaction immediately after it aborts. If a transaction aborts while trying to access a locked variable, the transaction waits until the lock is released before restarting.
- **Adaptive**: this strategy maintains multiple metadata like the number of restarts and a priority for each transaction to allow for a more elaborate CM implementation, upon conflict resolution the transaction with the lowest priority is aborted and after 4 restarts a transaction increases its priority to increase its chance of committing.

4 Performance Results

In this section, we show that the similarity between workload profiles helps choose a concurrency control that boosts performance and discard one that lowers performance.

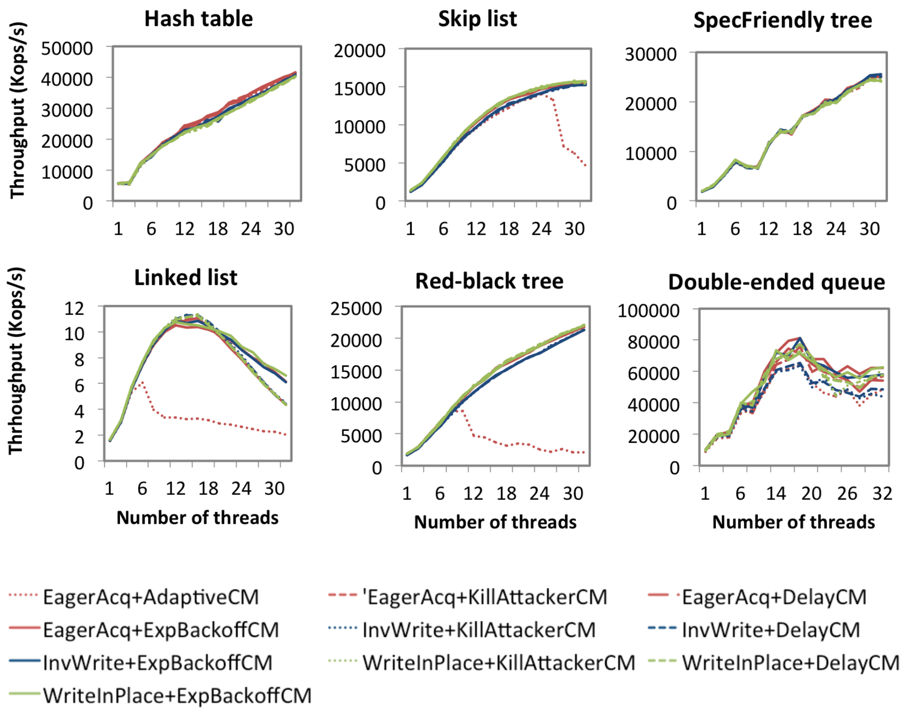

Fig. 1. The throughput of each concurrency control on various workloads when update rate is 10%

4.1 Experimental Settings

The machine is a 32-way x86-64 Intel Xeon E5-2450 machine with 2 sockets of 8 hyperthreaded cores each running at 2.1GHz Ubuntu 12.04.4 LTS and gcc 4.6.3. In all our experiments, we used TinySTM v1.0B [11] and Synchrobench v1.1.0-alpha [13] to average the value over 5 runs of 2 seconds each for each individual point with 1, 2, 4, 8, 12, 16, 18, 20, 22, 24, 26, 28, 30 and 32 threads and update ratios 10% and 50% effective (except for the double-ended queue whose operations are only updates). All data structures have expected size of 2^{16} during the experiments. The reason of our choices is that TinySTM offers contention management and conflict resolution policies that are common to most TM algorithms and Synchrobench is the most comprehensive benchmark-suite for evaluating synchronization techniques. The transactional memory (TM) algorithms include invisible write with eager acquirement or lazy acquirement and visible write with eager acquirement. The contention manager (CM) algorithms include the delay contention manager that waits until a lock is released before restarting the transaction that aborted due to its acquirement attempt, the strategy of killing the (attacker) transaction that detects the conflicts with another

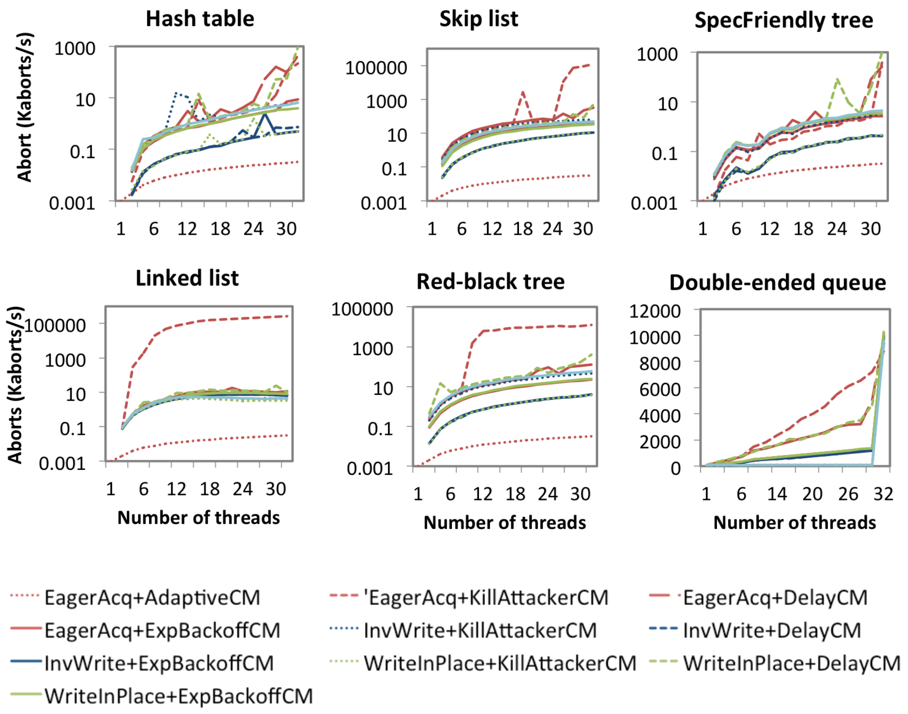

Fig. 2. The abort rate of each concurrency control on various workloads when update rate is 10%

(also called suicide), and the exponential backoff strategy that consists of waiting before restarting for a duration that increases exponentially each time the same transaction aborts.

4.2 Similar Profiles

Figure 1 gives the throughput as thousands of operations per second for each workload with 10% effective updates, meaning that 90% of the operations never modify the structure. The two workloads with the closest (most similar) profiles are the red-black tree and the speculation-friendly AVL tree as they have the same structure, however, the transactions execute differently on one or the other because rebalancing is not executed as frequently. The speculation-friendly tree has also shorter transactions and separate local rotation transactions that involve a constant number of nodes whereas the global red-black tree rotation is one unique transaction [5]. Due to this difference, the red-black tree throughput suffers dramatically more from the use of EagerAcq and adaptive CM than the speculation-friendly tree. Figure 2 gives thousands of aborted transactions per second in log scale for each workload with 10% of updates: the speculation-

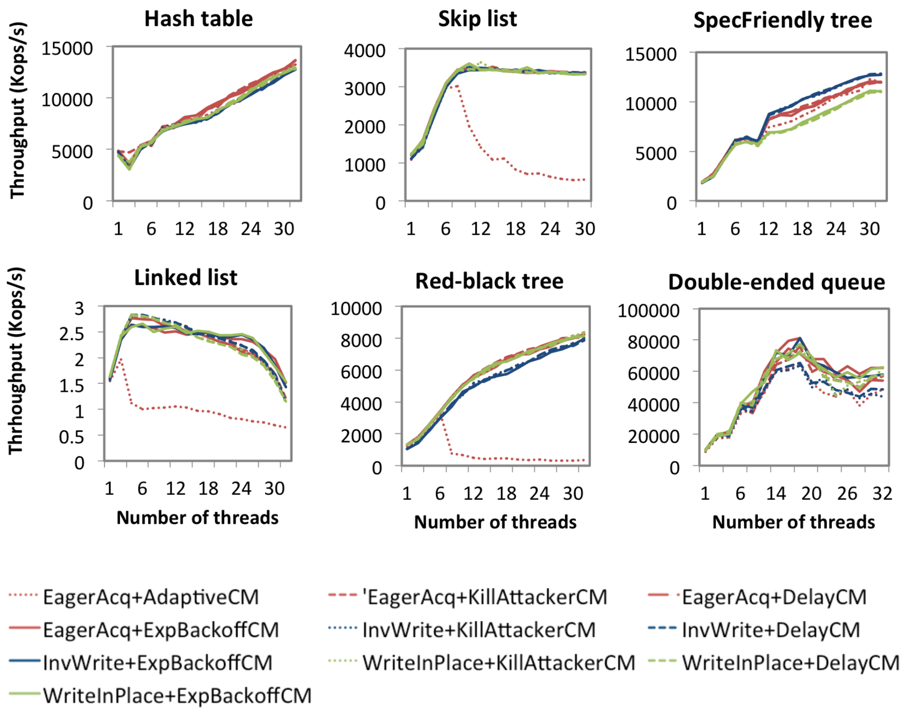

Fig. 3. The throughput of each concurrency control on various workloads when update rate is 50%

friendly tree does not abort as often as the red-black tree but we can see that WriteInPlace+Delay and EagerAcq+KillAttacker are combinations that trigger lots of abort on both structures.

Figure 3 and 4 give respectively the throughput and abort rate of each workload with 50% of updates. The skip list and the red-black tree, which have very close profile as well, experience very bad performance in the same scenario with a TM with eager acquirement and an adaptive CM. This phenomenon is exacerbated under higher contention as depicted in Figure 3. Actually, such a concurrency control prevents the red-black tree from scaling up to 12 threads (resp. 8 threads) at 10% updates (resp. 50% updates) while it prevents the skip list from scaling to 26 threads (resp. 10 threads) at 10% updates (resp. 80% updates). Note that this is not necessarily the case for other workloads, as the speculation-friendly AVL and hash table experience reasonable performance with such a combination of algorithms. Interestingly, the deque and the linked list experience bad performance with the same combination, but it is clear that the drop in performance is relatively more significant in the case of the red-black tree and the skip list than the other workloads.

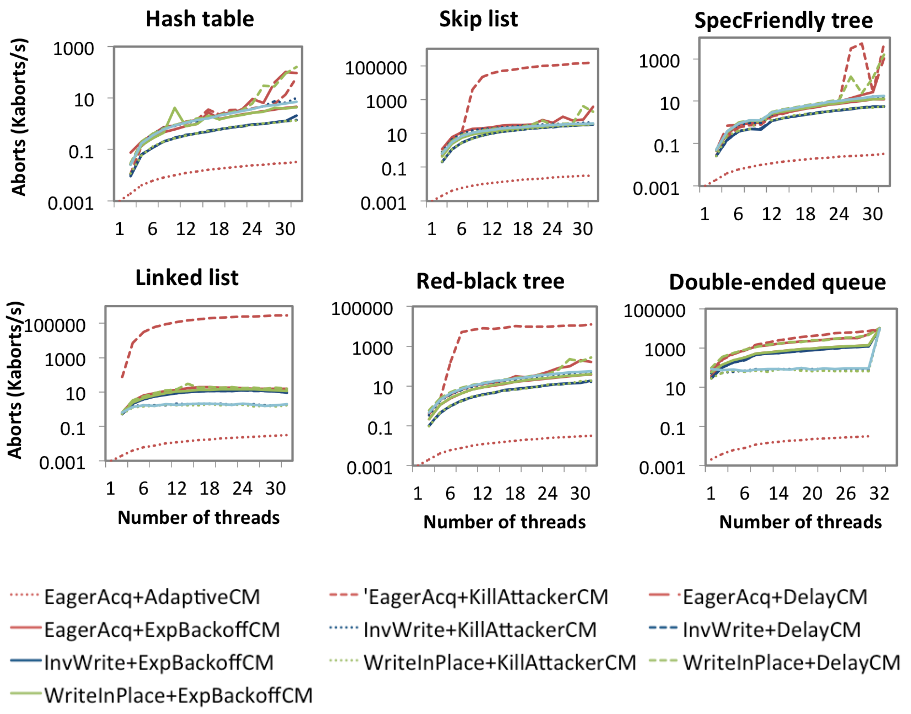

Fig. 4. The abort rate of each concurrency control on various workloads when update rate is 50%

We also observe that the best combination for the skip list and the red-black tree are similar: they tend to be boosted by the write-in-place and the invisible write transactional strategies almost irrespective of the contention manager used in combination. Moreover, besides the EagerAcq + AdaptiveCM combination we can clearly see that any of the remaining combinations leads to performance that are close to the peak performance of each of these two workloads, hence indicating that all could be chosen with negligible impact on the performance of these two workloads.

4.3 Different Profiles

The two workloads with the most different profiles, namely the linked list and the double-ended queue, experience very different performance results, which confirms the matching to their profile dissimilarities. An important remark is that our transactional algorithms are workload-oblivious, which is the reason why the linked list performance does not scale (we left the evaluation of more appropriate transactional algorithms, like Elastic [12] and Polymorphic transactions [15], to future work).

Workload	Hash table	Linked list	RB tree	AVL SF tree	Skip list	Deque
Best TM	EagerAcq	WriteInPlace	Inv. write	none	WrInPlace	WrInPlace
Worst TM	WriteInPlace	EagerAcq	EagerAcq	none	EagerAcq	InWrite
Best CM	ExpBackoff	KillAtt	Delay	ExpBackoff	KillAtt	ExpBackoff
Worst CM	Delay	Adaptive	Adaptive	Delay	Adaptive	KillAtt

Table 3. Empirical evaluation of the best and worst TM and CM algorithms for each workload

The first observation is that the overall performance of the linked list and double-ended queue (deque) is very different: the deque reaches the best peak performance while the linked list reaches the lowest performance of all workloads, which translates into a performance difference of up to 4 orders of magnitude between these two extremes. Note that the linked list implements a set whose operations may be read-only whereas the deque does not have read-only operations but the constant complexity of the deque pays off compared to the linear complexity of the linked list. Second, these two workloads do not maximize their performance with the same algorithm, in particular, the best performance of the linked list is reached when using the "Kill Attacker" contention manager, which is actually the one that minimizes the performance of the deque, as summarized in Table 3. Third, the linked list shows very distinct performance results depending on the algorithm used whereas the performance of the deque may perform reasonably well with any concurrency control combination.

Interestingly, the same algorithm (EagerAcq + AdaptiveCM) minimizes the performance of the linked list, the red-black tree and the skip list workloads, which are all reasonably close. We can conclude that the workloads with similar profiles tend to show similar performance when synchronized with the same concurrency control while it is generally not the case for workloads of substantially different profiles.

5 Discussion

Our preliminary results rely on static profiles that are computed prior to performance evaluation and allow us to classify the best and worst concurrency controls as summarized in Table 3. For this technique to be widely adopted, it is necessary to feed this profile database at runtime with new workloads that run for some time with randomly chosen concurrency controls. Once the profile of the application is refined, the concurrency control that benefits workloads with similar profiles is chosen to run the given workload. This process is restarted as the database of profile is fed with new workloads.

A dynamic profiling at runtime can help progressively picking the right concurrency control as the system learns about a growing amount of applications. While there exist complex applications where a pattern evolves during the application execution, we also notice that many applications use a recurrent workload pattern (e.g., the STAMP vacation application [2] would exclusively use the red-black tree structure).

An interesting direction to explore is thus to fully integrate our solution within the applications so that each application could dynamically switch between concurrency controls depending on its current execution pattern. Our previous work on transaction polymorphism already proposes compatible concurrency controls within a single transactional memory system [14,15]. This compatibility guarantees that transactions that abort can safely restart using a more appropriate concurrency control mechanism while others are concurrently running with the old concurrency control. This integration will facilitate the learning phase where information regarding performance and workload profile criteria would be collected. Finally, it would be ideal for our solution to automatically adjusts itself by adapting the weight of criteria based on past tests and observed performance results.

6 Related Work

Wang et al. exploited machine learning techniques to choose the best algorithms to execute a particular transactional workload [21]. They characterize the profiles of various workloads, including three data structure workloads, by approximating the length of transactional and non-transactional executions in clock ticks. Here we study six different data structures and we focus on transactional code, hence using shared accesses rather than clock ticks to measure an execution length.

Castro et al. used machine learning for binding threads to cores to optimize transactional applications but not to select the most suitable concurrency control algorithms [4]. Interestingly, they extended their work to adapt thread affinity at runtime and identified two workload characteristics in common with ours—transaction size and contention—but to identify the best affinity, not the best concurrency control algorithm [3].

Rughetti et al. implemented a technique to select the level of concurrency that maximizes the performance of various applications including STAMP ones [23,22]. Didona et al. [10] determine dynamically the optimal level of parallelism by exploiting online exploration in shared memory transactional applications and machine learning in distributed transactional applications. One could benefit from our technique to select the ideal concurrency control before using theirs to select the ideal level of concurrency.

Existing transactional memory algorithms feature various transaction algorithms one can select statically, including TinySTM [11], RSTM [19], $\mathcal{E}$-STM [12] to name a few. Our work is motivated by the well-known observation that transactional workloads are known to be highly dependent on the concurrency control used [2].

Collaborative filtering was initially used to compute the similarity between documents by Singhal [26] and was then applied to the context of data mining to compute the similarity within the same cluster by Tan et al. [27] while Boutet et al. [1] recently applied it to distributed recommendation systems. Lucia and

Ceze similarly exploited statistics in the Aviso framework [18] to predict the possibilities of bugs and to avoid appropriately the failures in future executions of the applications. Based on statistical inference Aviso schedules applications differently. The technique exploits collaboration among applications similarly to our approach but the goal of avoiding bugs differs from ours. As far we know, our work is the first to exploit recommendation systems in the context of concurrent applications.

7 Conclusion

Based on the similarities between transactional applications, we built a recommendation system to choose a concurrency control that maximizes performance depending on application similarities. Using 10 concurrency control mechanisms, our experiments show that the cosine similarity of 6 transactional workloads can be used to determine suitable concurrency controls. The similarities between a skip list and a red-black tree make their performance vulnerable to the same concurrency control algorithms. Finally, seemingly identical tree structures may have different profiles depending on their "speculation-friendliness", leading potentially to different performance with the same concurrency control.

Future work includes generalizing collaborative filtering on structures that use different synchronization primitives, including an efficient lock-based binary search tree [6] and CAS-based skip list [7]. The challenge here will be to propose the adequate synchronization techniques (e.g., lock, CAS and transaction) for an application based on its profile and the performance results observed with other applications.

References

1. Antoine Boutet, Davide Frey, Rachid Guerraoui, Arnaud Jégou, and Anne-Marie Kermarrec. WhatsUp Decentralized Instant News Recommender. In *IPDPS*, 2013.
2. Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, 2008.
3. Marcio Castro, Luis Fabricio Wanderley Goes, Luiz Gustavo Fernandes, and Jean-Francois Mehaut. Dynamic thread mapping based on machine learning for transactional memory applications. In *Euro-Par*, volume 7484 of *LNCS*, 2012.
4. Marcio Castro, Luis Fabricio Wanderley Goes, Christiane Pousa Ribeiro, Murray Cole, Marcelo Cintra, and Jean-Francois Mehaut. A machine learning-based approach for thread mapping on transactional memory applications. In *HIPC '11*, pages 1–10, 2011.
5. Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In *PPoPP*, pages 161–170, 2012.
6. Tyler Crain, Vincent Gramoli, and Michel Raynal. A contention-friendly binary search tree. In *Euro-Par*, volume 8097 of *LNCS*, pages 229–240, 2013.
7. Tyler Crain, Vincent Gramoli, and Michel Raynal. No hot spot non-blocking skip list. In *ICDCS*, Jul 2013.

8. Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*, pages 33–48, 2013.
9. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC*, volume 4167 of *LNCS*, pages 194–208, 2006.
10. Diego Didona, Pascal Felber, Derin Harmanci, Paolo Romano, and Jörg Schenker. Identifying the optimal level of parallelism in transactional memory applications. In *NETYS*, volume 7853 of *LNCS*, pages 233–247, 2013.
11. Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, pages 237–246, 2008.
12. Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In *DISC*, volume 5805 of *LNCS*, pages 93–107, 2009.
13. Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *PPoPP*, pages 1–10, 2015.
14. Vincent Gramoli and Rachid Guerraoui. Democratizing transactional programming. *Commun. ACM*, 57(1):86–93, Jan 2014.
15. Vincent Gramoli and Rachid Guerraoui. Reusable concurrent data types. In *ECOOP*, pages 182–206, Jul 2014.
16. Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC*, pages 258–264, 2005.
17. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.
18. Brandon Lucia and Luis Ceze. Cooperative empirical failure avoidance for multithreaded programs. In *ASPLOS*, pages 39–50, 2013.
19. Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. Technical Report 893, University of Rochester, Mar 2006.
20. Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, 2012.
21. Wang Q., Kulkarni S., Cavazos J., and Spear M. A transactional memory with automatic performance tuning. *ACM Transactions on Architecture and Code Optimization*, 8(4):1–23, 2012.
22. Diego Rughetti. *Autonomic Concurrency Regulation in Software Transactional Memories*. PhD thesis, Sapienza University of Rome, 2014.
23. Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia. Machine learning-based self-adjusting concurrency in software transactional memory systems. In *IEEE MASCOTS*, pages 278–285, 2012.
24. William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, pages 240–248, 2005.
25. William N. Scherer III and Michael L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, Jul 2004.
26. Amit Singhal. Modern information retrieval: a brief overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 24(4):35–42, 2001.
27. Pang-Ning Tan, Michael Steinbach, and Vipin Kumar Boston. *Introduction to Data Mining*. Pearson Addison Wesley, 2006.