Thread Affinity in Software Transactional Memory

Conference Paper · July 2020

DOI: 10.1109/ISPDC51135.2020.00033

CITATION
1

READS
31

4 authors:

Douglas Pereira Pasqualin
Universidade Federal de Pelotas
7 PUBLICATIONS **3** CITATIONS
SEE PROFILE

Matthias Diener
University of Illinois, Urbana-Champaign
67 PUBLICATIONS **511** CITATIONS
SEE PROFILE

André Rauber Du Bois
Universidade Federal de Pelotas
66 PUBLICATIONS **110** CITATIONS
SEE PROFILE

Maurício L. Pilla
Universidade Federal de Pelotas
98 PUBLICATIONS **176** CITATIONS
SEE PROFILE

Thread Affinity in Software Transactional Memory

Douglas Pereira Pasqualin*, Matthias Diener†,
André Rauber Du Bois* and Maurício Lima Pilla*‡

* *Computer Science Graduate Program (PPGC)*
Universidade Federal de Pelotas, RS, Brazil
{dp.pasqualin, dubois, pilla}@inf.ufpel.edu.br
† *University of Illinois at Urbana-Champaign, IL 61801, USA*
mdiener@illinois.edu
‡ *Google, Inc., Sunnyvale, CA 94089, USA*

***Abstract*—Software Transactional Memory (STM) is an abstraction to synchronize accesses to shared resources. It simplifies parallel programming by replacing the use of explicit locks and synchronization mechanisms with atomic blocks. A well-known approach to improve performance of STM applications is to serialize transactions to avoid conflicts using schedulers and mapping algorithms. However, in current architectures with complex memory hierarchies it is also important to consider where the memory of the program is allocated and how it is accessed. An important technique for improving memory locality is to map threads and data of an application based on their memory access behavior. This technique is called sharing-aware mapping. In this paper, we introduce a method to detect sharing behavior directly inside the STM library by tracking and analyzing how threads perform STM operations. This information is then used to perform an optimized mapping of the application's threads to cores in order to improve the efficiency of STM operations. Experimental results with the STAMP benchmarks show performance gains of up to 9.7x (1.4x on average), and a reduction of the number of aborts of up to 8.5x, compared to the Linux scheduler.**

***Index Terms*—Software Transactional Memory, Thread Mapping, Sharing-aware, Multicore**

I. Introduction

In parallel programming, mutual-exclusion locks are the most used abstractions for process synchronization. However, the semantics of locks is not intuitive. Developers need to explicitly acquire and release locks, making the source code hard to read and debug [1]. Also, locks have performance problems and can lead to deadlocks.

Transactional memory (TM) [2] is an abstraction to synchronize accesses to shared resources. It simplifies parallel programming by replacing the use of explicit locks with atomic blocks. By using TM, instead of explicit acquiring and releasing locks, the developer only needs to mark the block of code that he wants to be executed atomically as a transaction. The TM runtime is responsible to ensure a consistent execution, e.g., without deadlocks and race conditions. Although some CPUs have support for TM in hardware, software support is still necessary for larger transactions [1]. In this paper, we focus on software TM (STM), where transaction consistency is guaranteed by a software library.

A main concern of STM is the performance of transactions when there are concurrent accesses by multiple threads to the same data (conflicts). One technique to improve the performance of STM is the use of schedulers. A transactional scheduler acts proactively, using heuristics to prevent conflicts and to decide *when* and *where* a transaction should be executed [3]. The majority of STM schedulers try to avoid conflicts by serializing conflicting transactions.

In current computer architectures, there is an additional concern regarding complex memory hierarchies. In these systems, it is important to consider where the memory of the program is located and how it is accessed. Thus, the placement of threads and data is important to performance, improving locality of memory accesses. One technique for improving memory locality is to map threads and data of an application considering their memory access behavior. This technique is called *sharing-aware thread mapping* [4]. As STM is used to synchronize data accessed by multiple threads, an efficient thread mapping will help to make better usage of caches. Also, it is expected that thread mapping could help to reduce the number of aborts. As the shared resource is close in the underlying architecture, the transaction needs less time to complete the computation, reducing the time to conflict with others.

In this paper, we introduce a novel mechanism to perform sharing-aware thread mapping for STM applications. We analyze the application's sharing behavior inside the STM runtime, and use that information to calculate an optimized mapping of threads to cores. In contrast to previous proposals, our mechanism can perform more accurate mapping decisions since the STM runtime has precise information about shared data and the threads that access it. Furthermore, compared to techniques that rely on memory access tracing, our proposal has a much lower overhead since it can focus only on memory accesses to data that is actually shared. Moreover, there is no need to change the application and the developer does not need to be aware of the underlying hardware or the memory access behavior.

The rest of this paper is organized as follows. The next

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 and PROCAD/LEAPaD

Section presents the main concepts of STM and sharing-aware mapping. Section III describes our proposed mechanism to discover the communication behavior of an STM application. Experimental results are presented in Section IV. Finally, Section V discusses related work and the next Section concludes the paper.

II. Background: Sharing-aware mapping in STM

This section presents a brief overview of software transactional memory and sharing-aware mapping. We also discuss the potential improvements of such a mapping in STM using a simple Array Sum benchmark.

A. Software Transactional Memory

Software transactional memory (STM) is an abstraction to synchronize accesses to shared resources. Instead of using locks, the programmer only needs to enclose the critical section in an atomic block, which will be executed as a transaction. The concept of transactions was borrowed from Database systems. A transaction *commits* if executed without conflicts, hence all operations and results are made visible to the rest of the system. If conflicts are detected, a transaction *aborts*, i.e., all operations are discarded, and the transaction needs to restart until a commit is possible. Although the main purpose of STM is to provide a simple interface to manage access to shared resources, its implementation is not trivial. Many different design options are available like transaction granularity, version management, conflict detection and resolution.

Granularity is the dimension used for conflict detection, i.e., the level used for keeping track of memory locations. The most common options are memory *word* or *object*. Version management is used to manage the tentative writes of concurrent transactions. The options are *eager*, where data is modified directly in memory or *lazy*, where data is updated in memory during the commit phase. Conflict detection has the same names and intuitions of version management. Another important component is the contention manager (CM). When two transactions conflict, the transaction that detected the conflict, asks the CM what to do. The actions could be to abort immediately or wait for a determined time, allowing the other transaction to finish, or force an abort of the enemy transaction [5]. `TinySTM` [6] is one of the most used STM libraries and still is considered a state-of-art STM implementation [7], [8]. `TinySTM` uses word granularity and has configurable version management and CMs.

B. Sharing-aware mapping

An important way to improve the locality and performance in multicore and NUMA systems is to schedule threads and data according to their *memory access behavior* [4]. In this work, the focus is on the schedule of threads. Thus, thread mapping associates threads to cores, improving the cache usage and interconnections, i.e., threads are mapped to cores that are close to each other in the underlying architecture. A mechanism to schedule threads based on the memory access behavior is called *sharing-aware thread mapping* [4]. Although most OS have routines to deal with thread mapping, they generally do not take memory access patterns into consideration. The scheduler *Completely Fair Scheduler* (CFS) [9] used by default in the Linux kernel mainly focuses on load balancing. To determine thread mapping based on memory access, it is necessary to know how threads share data [10]. This information is usually represented as a *communication matrix*. Together with information about the hardware hierarchy, a mapping algorithm uses the communication matrix to calculate an improved mapping of threads to cores.

III. A mechanism for sharing-aware mapping in STM

This section presents our proposed communication detection and mapping mechanism for STM applications.

A. Detecting communication in STM applications

The first step to detect the communication and memory access behavior of applications is to know which memory addresses are being accessed. In shared memory, communication is implicit, i.e., it is performed through accesses to shared memory areas [10]. One advantage of STM implementations is that on each transactional data access operation, the memory location is explicitly included as a parameter of the operation, and is available to the STM runtime.

Thus, it is not necessary to instrument the application using external tools to extract sharing information. Detecting communication behavior accurately in parallel applications often has a high overhead [10], for instance by using a memory tracer such as `numalize` [11]. In STM applications however, we are interested in transactional variables only. Since the STM runtime has precise information about shared variables and running threads, it is possible to detect the communication, determining which threads have accessed the same transactional variable. When 2 distinct threads have accessed the same address a communication event between them is updated. The amount of communication between threads is stored in a communication matrix. This matrix represents the amount of communication between pairs of threads and is used by mapping algorithms to calculate optimized placements.

Our algorithm to detect the communication works as follows. On each transactional data access operation, the accessed memory address is kept in a hash table (Fig.1, line 8), whose key is the memory address. Each position of the hash table contains a data structure (Fig.1, line 1) with the memory address and the last 2 threads that have accessed it. For improved accuracy, if a conflict occurs in the hash table, a linked list with all memory addresses with the same hash is kept (Fig.. 1, line 5). The algorithm for detecting the communication pattern is shown in Fig.1. In each transactional read or write, the function `check_comm` is called (line 16). This function receives the address being accessed and the identifier of the thread that is accessing it. In line 17, `getAddressEntry` is a function that returns the information of the memory access being accessed, i.e, it looks up the address in the hash table.

```
1  typedef struct address_entry_t {
2      unsigned long int address;
3      unsigned int share1;
4      unsigned int share2;
5      struct address_entry_t* next;
6  } address_entry;
7
8  address_entry hash_table[HASH_TABLE_SIZE];
9  unsigned int comm_matrix[NUM_THREADS][NUM_THREADS];
10
11 void update_comm_matrix(int thread1, int thread2) {
12   comm_matrix[thread1][thread2]++; /*symmetric matrix*/
13   comm_matrix[thread2][thread1]++; /*symmetric matrix*/
14 }
15
16 void check_comm(unsigned long int addr, int threadId) {
17   address_entry *elem = getAddressEntry(addr);
18   int accesses = count_accesses(elem);
19   switch (accesses) {
20     case 0: /* first access to the address */
21       elem->share1 = threadId;
22       break;
23     case 1: /* one previous access */
24       if (elem->share1 != threadId) {
25         update_comm_matrix(threadId, elem->share1);
26         elem->share2 = elem->share1;
27         elem->share1 = threadId;
28        }
29       break;
30     case 2: /* two accesses, push share1 position */
31       if (elem->share1 != threadId &&
32               elem->share2 != threadId) {
33         update_comm_matrix(threadId, elem->share1);
34         update_comm_matrix(threadId, elem->share2);
35         elem->share2 = elem->share1;
36         elem->share1 = threadId;
37       } else if (elem->share1 == threadId) {
38         update_comm_matrix(threadId, elem->share2);
39       } else if (elem->share2 == threadId) {
40         update_comm_matrix(threadId, elem->share1);
41         elem->share2 = elem->share1;
42         elem->share1 = threadId;
43       }
44       break;
45     }
46 }
```

Fig. 1: Source code of the mechanism for detecting communication patterns.

Thus, on line 18 we count how many accesses this address had before the current access. If it had one previous access (line 23) and the current thread accessing it (`threadId`) is different from the previous (line 24) a communication event is found. Hence, the communication matrix is updated (line 25), calling the function `update_comm_matrix` (line 11), which increments the amount of communication between the two threads by one. The communication matrix is square and symmetric because the amount of communication, for instance, between threads 2 and 3 is the same from 3 and 2 (lines 12 and 13). Also, the matrix has an order equal to the maximum number of threads. A final case is if the address had 2 accesses before the current one (line 30). In lines 31 and 32, we verify if the current thread accessing the address (`threadId`) is different from the 2 previous accesses. If true, a communication event between the current thread and the previous share is updated (lines 33 and 34). After that, the oldest access (`share2`) is removed (line 35). However, the current thread accessing the address could be the same from a previous access (lines 37 and 39). In that case, we need to update the communication event correctly, and remove the old access, if necessary (lines 41 and 42).

B. Calculating the mapping

The communication matrix generated by the proposed mechanism is sent to `EagerMap` [12], a task mapping algorithm for communication-aware mapping, to generate the thread mapping. It is worth noting that other task mapping algorithm could be used in this step. However, as presented in [12], `EagerMap` was faster and generated a more efficient mapping. In this work, we use a *static* approach, i.e., an application needs a prior execution to generate a communication matrix with their memory access behavior. Using the generated mapping, the application can be re-executed, pinning threads to cores using the function `pthread_setaffinity_np`.

C. Implementation

We implemented our proposed mechanism inside the state-of-art STM library `TinySTM` [6], version 1.0.5. The majority of the modifications were made in the file `stm_internal.h`. The function `check_comm` (Fig.1, line 8) is called inside the functions `stm_write` and `stm_load` from `TinySTM`.

IV. Evaluation

A. Methodology

In order to evaluate our proposal, we used our modified version of the `TinySTM` library (Sect. III-C), with the default configuration: *lazy* version management, *eager* conflict detection and CM *suicide*.

The applications used in the experiments were all eight benchmarks from the Stanford Transactional Applications for Multi-Processing (`STAMP`) [13] version 0.9.10, and two micro-benchmarks (HashMap and Red-Black tree) from [14]. `STAMP` is a collection of realistic workloads, covering a wide range of transactional execution cases. Applications were executed using 8, 16, 32 and 64 threads, on an **Opteron** machine with four AMD 6276 processors and 128 GiB of RAM running Linux kernel 3.13. Each socket has eight 2-SMT cores, totaling 64 threads. Each CPU has 2 memory controllers (for a total of 8 NUMA nodes), and 16× 32 KB L1d, 8× 64 KB L1i, 8× 2 MB L2 and 2× 6 MB L3 caches.

For comparing our proposed approach, each application ran ten times, using the default Linux scheduler, i.e., without any intervention, the generated mapping from the communication matrices using `EagerMap` [12], which we call *sharing-aware* and, the *compact*, *scatter* and *round-robin* mappings. Compact places threads on sibling cores, sharing all levels of cache, reducing data access latency [15]. This approach is useful where threads communicate often with their neighbors [16]. Scatter distributes threads across different processors, avoiding cache sharing, thus, reducing memory contention [15] and balancing the computation load. Finally, round-robin is a mix between compact and scatter, where only the last level of cache is shared [15]. The compact, scatter, and round-robin mappings were calculated using the `hwloc` library [17]. The overhead added to calculate the mapping is approximated 70 ms. The mapping is applied when a thread calls the `stm_init_thread` function of the `TinySTM`

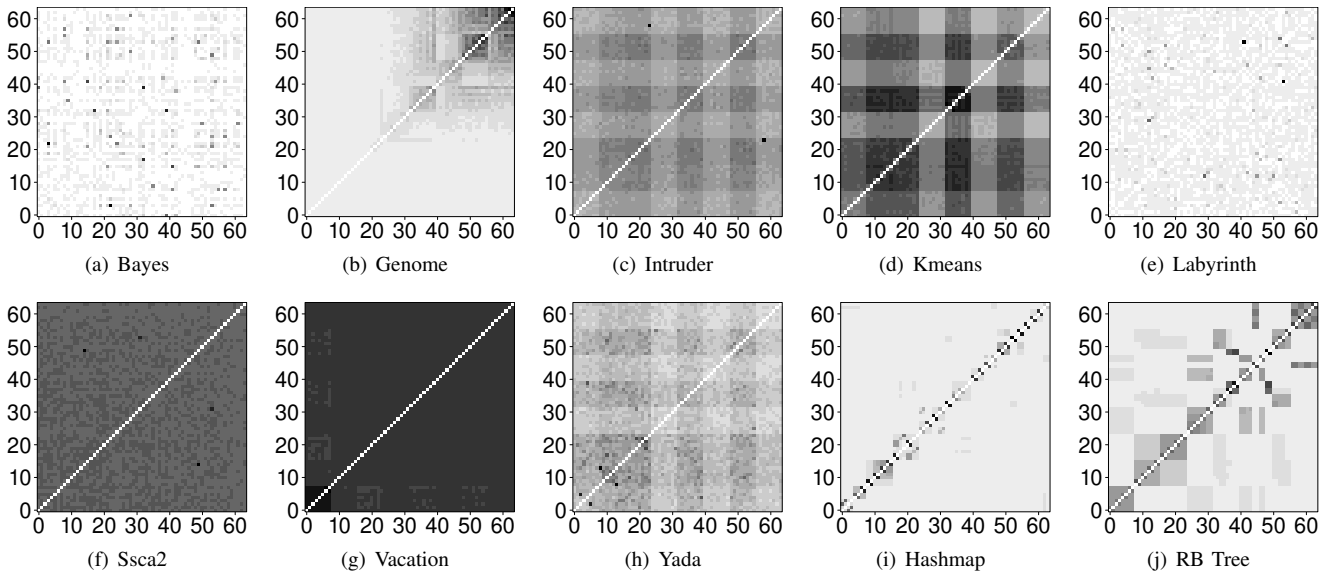


Fig. 2: Communication matrices detected by our mechanism for 64 threads.

library, which informs the runtime that the thread will perform transactional operations.

B. Results

Fig. 2 shows the generated communication matrices, for 64 threads, using our mechanism described in Sect. III, whereas Fig. 3 presents the performance results. In Fig. 2, darker cells indicate more communication between the pair of threads. In Fig. 3, each row contains the application, the execution time (in seconds), the number of aborts and the speedup over the Linux scheduler.

Bayes is an application with high contention, spending lots of time inside transactions [13]. Sharing-aware was the best mapping in 8 and 16 threads with a speedup of 1.7× and 1.5× respectively, and the second place using 32 threads. The Linux mapping generates a lot of aborts using 32 and 64 threads. However, aborts do not hurt the performance in this benchmark. In fact, Linux was the best mapping for 64 threads, despite many aborts. Although the number of aborts could be a metric to measure the performance (less wasted work), sometimes the performance is not directly related with them [18].

Genome has low contention and spends lots of time inside transactions [13]. Sharing-aware mapping had a median result on 8 and 16 threads. Analyzing the scenario with 64 threads, even though threads with IDs higher than 50 have a well-defined pattern (Fig.2(b)), it was not enough to put them closer to increase the performance. Scatter had a better speedup in all scenarios, even with the higher number of aborts using 64 threads.

Intruder has high contention and spends medium time inside transactions [13]. Sharing-aware was the best mapping in 64 threads and the second place in 16 and 32 threads. This behavior was similar in aborts. The communication pattern (Fig.2(d)) indicates that in this application threads communicate often with their neighbors. For this reason, Compact achieved good results.

Kmeans has low contention and spends little time inside transactions [13]. This application behaves similarly to *Intruder*, including the results of the mappings. For 64 threads, sharing-aware achieved a speed of 1.2× compared to Linux.

Labyrinth has high contention and spends lots of time inside transactions [13]. Sharing-aware achieved the best results under 8 and 64 threads. However, with less than 64 threads the results of all mappings were very similar. This result can be explained because this application has a communication pattern where threads have balanced communication between others (Fig.2(e)).

Ssca2 has low contention, spending little time inside transactions [13]. Sharing-aware was the best mapping with 32 threads and the second one using 16 and 64 threads. Regarding aborts, Linux generated a higher number of aborts with 32 and 64 threads.

Vacation has medium contention and spends high time inside transactions [13]. Analyzing the speedup, all mappings have similar results. However, sharing-aware was the best under 64 threads, with a speedup of 9.7×. Linux increases in about 4 times the execution time and aborts under 64 threads.

Yada has medium contention and spends high time inside transactions [13]. Regarding the Linux scheduler, this application has a similar behavior as *Vacation*. However, in all the numbers of threads, Linux performance was poor, including aborts, and all other mappings are similar. The lack of distinct results (excluding Linux) can be explained by the low execution time of the application.

The last two micro-benchmarks, *Hashmap* and *Red-black tree* have a very similar communication pattern (Fig.2(i) and 2(j)). Like *Intruder*, those applications communicate often be-

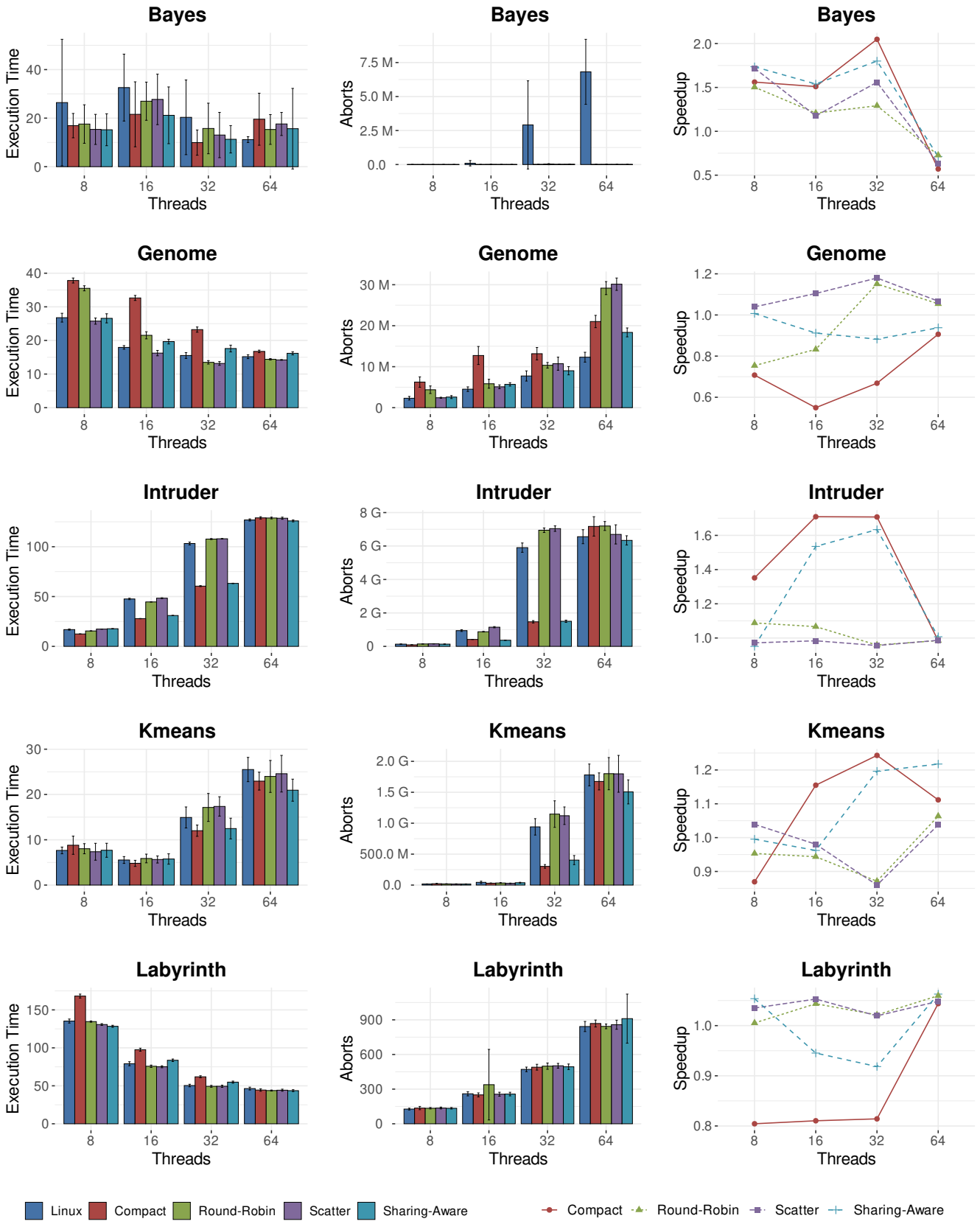


Fig. 3: Execution time, number of aborts and speedup.

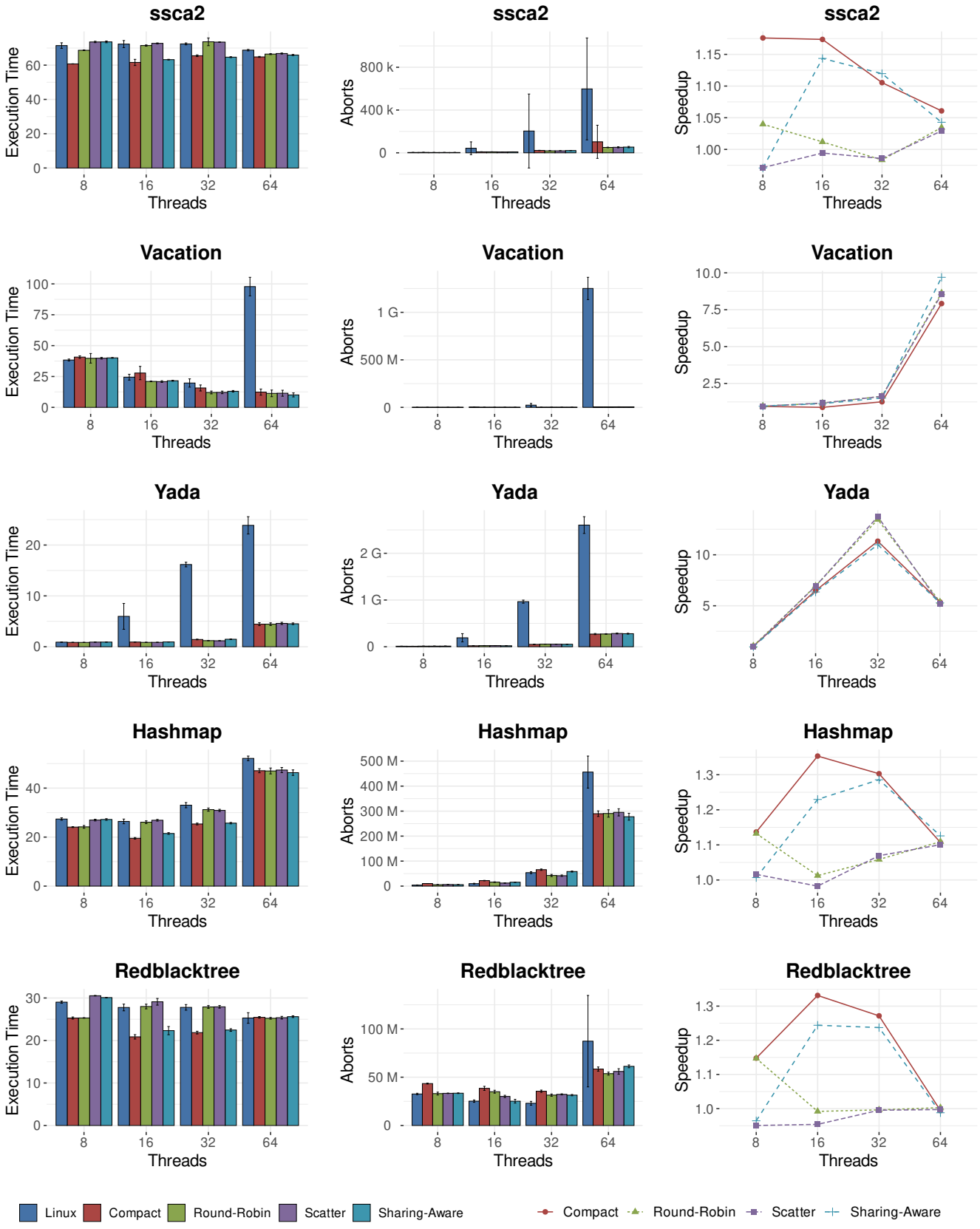
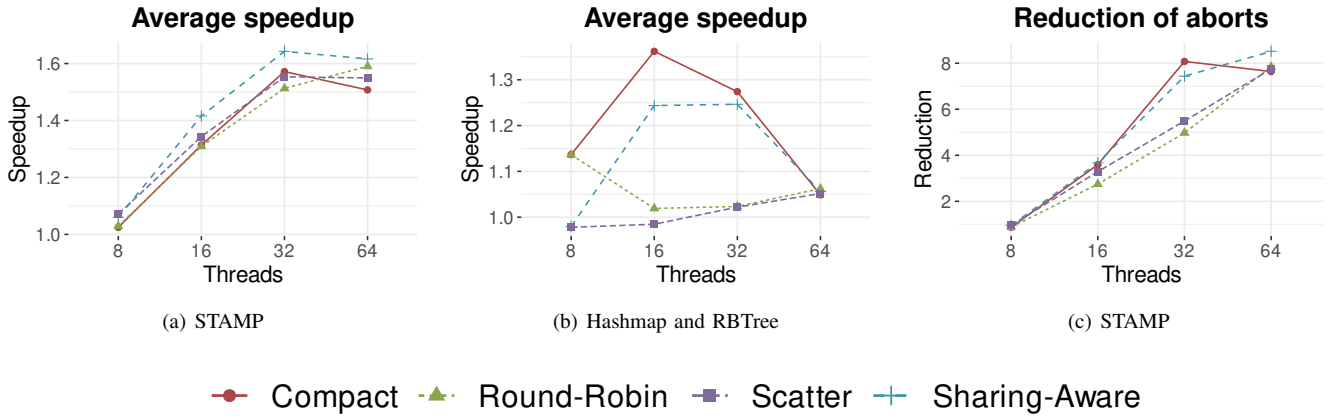


Fig. 3: Execution time, number of aborts and speedup (cont.).



(a) STAMP (b) Hashmap and RBTree (c) STAMP

Fig. 4: Average speedup and reduction of aborts.

tween neighboring threads. For this reason, Compact achieved good results. However, in *Hashmap* sharing-aware achieved the best result under 64 threads, both on execution time and number of aborts.

Analyzing all 10 applications and 4 thread configurations, there are 40 scenarios. Compact was the best in 16 of them. On the other hand, it was the worst in 14. Scatter was the best in 8 and worst in 14. Sharing-aware was the best in 9, the worst in 6 and the second place in 15 of them. The best case was on *Vacation*, with speedup of 9.7×. Besides, it had the best speedup over all `STAMP` benchmarks (Fig.4(a)), with gains of up to 1.6x using 32 threads, and 1.4x on average, compared to Linux. Round-Robin was the worst mapping, being the best only in 4 scenarios. Regarding aborts, sharing-aware reduced the number of aborts of up to 8.5× using 64 threads, compared to the Linux scheduler (Fig.4(c)). It is worth noting that only transactional operations were tracked to extract memory access behavior. The intuition is that if a memory location is shared by more than one thread, it should be protected using a transactional operation (read or write). However, a potential disadvantage of this approach compared to full memory tracing with tools such as `numalize` [11] could be the lack of information regarding false sharing on cache lines. Nevertheless, the results presented in this section indicate that the STM system has enough information to generate an accurate communication pattern, increasing the performance, mainly over the Linux scheduler. In fact, Linux was the best scheduler only in 3 scenarios: *Vacation* using 8 threads and *Bayes* and *Red-Black tree* using 64 threads. Overall, sharing-aware mapping achieved the best results.

The overhead to detect the memory access behavior depends on the amount of shared data accessed. Some applications have almost no overhead, and in the worst case, the execution time was increased by 17×. For comparison, in HashMap and Red-Black tree the overhead was 3× and 4×, respectively, whereas `numalize` [11] has an overhead of 18× and 17×, respectively, for these applications. As the detection and calculation of the mapping is done before the actual application execution, there is no further runtime overhead.

V. Related Work

Scheduling is a well-known technique for improving the performance of STM. However, the main objective of transactional schedulers is serializing transactions when conflicts are detected or predicted. Adaptive transaction scheduling (ATS) was one of the first works to propose a transaction scheduler [19]. It tries to predict future conflicts between transactions, using a metric called *contention intensity*. Scheduling techniques in STM were widely explored in the literature, for instance, in [3] and [20]. A survey summarizing scheduling techniques for STM is presented in [7].

Another idea is to use concurrency control, i.e., limit the total number of threads executing concurrently. They rely on the idea that an excessive number of threads can hurt the performance in a high contention environment, mainly due to a higher number of aborts. Adaptive Concurrency Control (ACC) [21] was one of the first works to propose this technique. The concurrency control mechanism was also studied in [22] and [23].

Few works use scheduling techniques, more specifically thread mapping, for better usage of machine resources. In [24], concurrency control and an affinity-aware thread migration were proposed. The idea is to reducing cross-die cache invalidation, keeping transactions that conflict often in sibling cores to share caches. When a transaction aborts, it detects the enemy transaction, and the likelihood of conflict between them is updated in a matrix. In a predefined amount of time, a pair of threads is chosen to migrate between sockets. This approach is focused on machines with dual-processors. Also, the sharing-aware approach presented in this paper is more accurate, because it is based on each transactional read and write, besides the thread mapping is global, i.e., taking into consideration all threads.

Castro et al. [15] use the abort ratio of transactional applications to decide the thread mapping strategy. According to the abort ratio and a given threshold, they dynamically change

the thread mapping between scatter, compact or round-robin. The main objective of [25] is to define dynamically the best number of thread allowed to run concurrently, maximizing the throughput. When the number of threads is less than half of the total number of cores, another mechanism is triggered, changing the thread mapping to round-robin. According to the throughput achieved, the mapping could be back to Linux default or changed to compact or scatter. In both proposals, the memory access behavior of the applications was not taken into consideration.

Góes et al. [26] proposed a mechanism to improve memory affinity of STM applications that exhibit a specific worklist pattern. To improve memory affinity, their mechanism relies on the use of page allocation policies (bind or cyclic) and data prefetching by using helper threads. To use the proposed mechanism, the STM application must be adapted to use the API of the `OpenSkel` framework, which implements the proposed mechanism.

VI. Conclusion

Sharing-aware mapping is a well-known technique for improving the performance of applications. It maps threads and data considering their memory access behavior, making the communication and cache usage more efficient. In this paper, a mechanism to extract the memory access behavior of STM applications is proposed. The main advantage is that the applications are not modified, only the STM system. Although only transactional operations are tracked to extract the memory access behavior, the experimental results show that only the information inside the transactional memory system is enough to improve the performance of STM applications. For the `STAMP` benchmarks, we achieved a speedup of up to 9.7× (1.4× on average), and a reduction of the number of aborts of up to 8.5× (3.8× on average), compared to the Linux scheduler. For the future, we intend to study how to implement the sharing-aware thread mapping dynamically. The main advantage is that a preliminary run to get the sharing information will not be necessary. Also, we will explore sharing-aware data mapping.

References

[1] G. Anthes, "Researchers simplify parallel programming," *Commun. ACM*, vol. 57, no. 11, pp. 13–15, Oct. 2014.

[2] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory*, 2nd ed. San Rafael, California, USA: Morgan and Claypool Publishers, 2010.

[3] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh, "Preventing versus curing: Avoiding conflicts in transactional memories," in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2009, pp. 7–16.

[4] E. H. M. Cruz, M. Diener, and P. O. A. Navaux, *Thread and Data Mapping for Multicore Systems*. Cham, Switzerland: Springer International Publishing, 2018.

[5] R. Guerraoui, M. Herlihy, and B. Pochon, "Towards a theory of transactional contention managers," in *Proceedings of the 25th Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2006, pp. 316–317.

[6] P. Felber, C. Fetzer, T. Riegel, and P. Marlier, "Time-based software transactional memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, pp. 1793–1807, 2010.

[7] P. D. Sanzo, "Analysis, classification and comparison of scheduling techniques for software transactional memories," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 12, pp. 3356–3373, dec 2017.

[8] P. Poudel and G. Sharma, "Adaptive versioning in transactional memories," in *Stabilization, Safety, and Security of Distributed Systems*, M. Ghaffari, M. Nesterenko, S. Tixeuil, S. Tucci, and Y. Yamauchi, Eds. Cham: Springer International Publishing, 2019, pp. 277–295.

[9] C. S. Wong, I. Tan, R. D. Kumari, and F. Wey, "Towards achieving fairness in the Linux scheduler," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 34–43, Jul. 2008.

[10] M. Diener, E. H. M. Cruz, M. A. Z. Alves, and P. O. A. Navaux, "Communication in shared memory: Concepts, definitions, and efficient detection," in *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb 2016, pp. 151–158.

[11] M. Diener, E. H. Cruz, L. L. Pilla, F. Dupros, and P. O. Navaux, "Characterizing communication and page usage of parallel applications for thread and data mapping," *Performance Evaluation*, vol. 88-89, pp. 18–36, jun 2015.

[12] E. H. M. Cruz, M. Diener, L. L. Pilla, and P. O. A. Navaux, "EagerMap: A task mapping algorithm to improve communication and load balancing in clusters of multicore systems," *ACM Trans. Parallel Comput.*, vol. 5, no. 4, Mar. 2019.

[13] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *IEEE International Symposium on Workload Characterization*. IEEE CS, Sept 2008, pp. 35–46.

[14] N. Diegues, P. Romano, and L. Rodrigues, "Virtues and limitations of commodity hardware transactional memory," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Aug. 2014, pp. 3–14.

[15] M. Castro, L. F. W. Góes, and J.-F. Méhaut, "Adaptive thread mapping strategies for transactional memory applications," *Journal of Parallel and Distributed Computing*, vol. 74, no. 9, pp. 2845 – 2859, 2014.

[16] P. N. Soomro, M. A. Sasongko, and D. Unat, "BindMe: A thread binding library with advanced mapping algorithms," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 21, p. e4692, 2018.

[17] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in HPC applications," in *18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE CS, Feb 2010, pp. 180–186.

[18] D. Dice and N. Shavit, "Understanding tradeoffs in software transactional memory," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 21–33.

[19] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: ACM, 2008, pp. 169–178.

[20] H. Rito and J. Cachopo, "Adaptive transaction scheduling for mixed transactional workloads," *Parallel Comput.*, vol. 41, no. C, pp. 31–49, Jan. 2015.

[21] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson, "Advanced concurrency control for transactional memory using transaction commit rate," in *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 719–728.

[22] K. Chan, K. T. Lam, and C.-L. Wang, "Adaptive thread scheduling techniques for improving scalability of software transactional memory," in *10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN 2011)*. Innsbruck, Austria: ACTA-Press, February 2011, pp. 91–98.

[23] K. Ravichandran and S. Pande, "F2C2-STM: Flux-based feedback-driven concurrency control for STMs," in *IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 927–938.

[24] K. Chan, K. T. Lam, and C. L. Wang, "Cache affinity optimization techniques for scaling software transactional memory systems on multi-CMP architectures," in *14th Int. Symposium on Parallel and Distrib. Comput.* IEEE CS, June 2015, pp. 56–65.

[25] N. Zhou, G. Delaval, B. Robu, E. Rutten, and J.-F. Méhaut, "An autonomic-computing approach on mapping threads to multi-cores for software transactional memory," *Concurrency Computat Pract Exper.*, vol. 30, no. 18, p. e4506, may 2018.

[26] L. F. Góes, C. P. Ribeiro, M. Castro, J.-F. Méhaut, M. Cole, and M. Cintra, "Automatic skeleton-driven memory affinity for transactional worklist applications," *Int. J. Parallel Program.*, vol. 42, no. 2, p. 365–382, Apr. 2014.