THREAD AND DATA MAPPING IN SOFTWARE TRANSACTIONAL MEMORY: AN OVERVIEW

Douglas Pereira Pasqualin
Computer Science Graduate Program (PPGC)
Universidade Federal de Pelotas
Rio Grande do Sul, 96075-630, Brazil
dp.pasqualin@inf.ufpel.edu.br

Matthias Diener
University of Illinois Urbana-Champaign
Illinois, 61801, USA
mdiener@illinois.edu

André Rauber Du Bois
Computer Science Graduate Program (PPGC)
Universidade Federal de Pelotas
Rio Grande do Sul, 96075-630, Brazil
dubois@inf.ufpel.edu.br

Maurício Lima Pilla
Computer Science Graduate Program (PPGC)
Universidade Federal de Pelotas
Rio Grande do Sul, 96075-630, Brazil
pilla@inf.ufpel.edu.br

June 6, 2022

ABSTRACT

In current microarchitectures, due to the complex memory hierarchies and different latencies on memory accesses, thread and data mapping are important issues to improve application performance. Software transactional memory (STM) is an abstraction used for thread synchronization, replacing the use of locks in parallel programming. Regarding thread and data mapping, STM presents new challenges and mapping opportunities, since (1) STM can use different conflict detection and resolution strategies, making the behavior of the application less predictable and; (2) the STM runtime has precise information about shared data and the intensity with each thread accesses them. These unique characteristics provide many opportunities for low-overhead, but precise statistics to guide mapping strategies for STM applications. The main objective of this paper is to survey the existing work about thread and data mapping that uses solely information gathered from the STM runtime to guide thread and data mapping decisions. We also discuss future research directions within this research area.

Keywords Software Transactional Memory · Thread Mapping · Data Mapping · Communication

1 Introduction

Since year 2000, multicore processors have become the main configuration of new processor developments. This decision was taken due to microarchitectural limitations, higher power consumption and, heat dissipation involved in improving the performance of a single CPU core [47]. Since then, the number of cores in a single chip has been growing every year. Besides, servers normally have multiple multicore processors, where each socket is connected directly to a *local* memory module [24]. These architectures are called NUMA (Non-Uniform Memory Access) systems and are becoming dominant in servers [5]. Accessing data that is mapped to a memory module that belongs to a different processor, that is, in a *remote node*, implies a higher latency, making the access time non-uniform, that is, depending on the location of the data.

Due to multicore processors being widely available, parallel programming is becoming even more important. A crucial issue that arises in parallel programming is thread synchronization; locks are still the most used abstraction for this purpose. However, locks often make the source code hard to read and debug, leading to problems such as deadlocks [29, 3]. An alternative abstraction to replace mutual-exclusion locks in parallel programming is *Transac-*

tional Memory (TM) [27], in which critical-sections are accessed using transactions similar to the ones available in databases. The TM runtime is responsible for ensuring a consistent execution, for example, without deadlocks and race conditions. A transaction that has executed without conflicts can commit, that is, update the memory with the new values. If a conflict is detected, an abort is executed and a transaction is reinitialized until a commit is possible. TMs can be implemented in hardware (HTM), software (STM), or both (hybrid). HTM has the advantage of a lower overhead compared to STM since they do not need instrumentation to track transactional operations [17]. However, HTM has resource limitations. For example, when the footprint of a transaction exceeds the L1 cache capacity, they are aborted [18]. In that case, software alternatives are necessary to guarantee progress. This paper focuses on STM implementations since all prior proposals found in the literature are based on STM.

Another important issue in multicore processors is the mapping of thread and data. Due to complex memory hierarchies and different latencies in memory accesses on these processors, a thread and data placement that improves the use of memory controllers and data locality is important to achieve good performance. Although thread and data mapping is widely studied for general purpose parallel applications [19] there is a lack of research in the STM domain. More specifically about thread mapping, STM adds new challenges such as different kinds of conflict detection and resolution. Hence, the best thread mapping depends on the STM configuration [9]. Also, the STM runtime has precise information about memory areas that are shared between threads, their respective memory addresses, and the intensity with which they are accessed by each thread, providing interesting mapping opportunities [40]. In that case, a thread mapping based only on STM accesses will have a lower overhead than other proposals that focus on general applications since it is not necessary to keep track of all memory access of the applications to determine the mapping.

In this paper, we describe and survey proposals that focus on using the information provided by the STM runtime to guide thread and data mapping. Previous existing surveys focused only on general applications [19]. To the best of our knowledge, there are no prior surveys of thread and data mapping in the context of STM. We found that the STM runtime provides accurate information about the sharing behavior of the application being able to provide support information for an improved thread mapping. For data mapping, our survey showed that it might be unfeasible to rely only on the information provided by the STM runtime to perform an efficient data mapping since the STM runtime usually has access to only a fraction of the entire memory accessed by the application, limiting the accuracy of a data mapping algorithm.

The rest of this paper is organized as follows. The next section presents the main concepts of STM and thread and data mapping. Section 3 surveys and discusses the related work on thread and data mapping in the context of STM. We also shown a table comparing and classifying all discussed works. In Section 4 we propose some research directions to fill gaps missing in the related work. Finally, Section 5 concludes the paper.

2 Background

2.1 Software Transactional Memory

Transactional memory (TM) is an abstraction to synchronize accesses to shared variables. 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 Databases. The first TM implementation purely on software (STM) was proposed by Shavit and Touitou [45]. The execution of a transaction needs to be *atomic*. Atomicity requires that a transaction is executed as a whole or it needs to appear as it was never executed [27]. 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. This idea is associated with another important property called *isolation*: all memory updates of a running transaction can not be visible to other transactions before a commit.

Although the main purpose of STM is to provide a simple interface to manage access to shared data, its implementation is not trivial. Many different design options are available such as transaction granularity, version management, conflict detection, and resolution. The *granularity* is the dimension used for conflict detection, i.e., the level used for keeping track of memory locations. For STM, the most used options are **word** or **object**. *Version management* manages the writes of concurrent transactions to memory locations. It can be **eager**, where data is modified directly in memory and an *undo-log* is used to restore old values in case of aborts; or **lazy**, where data is updated in a *redo-log*. During the commit, the log is used to set new values to memory. *Conflict Detection* uses the same nomenclature as version management. In **eager** conflict detection, conflicts are verified on each memory location accessed. To access a value, a transaction needs to acquire its ownership. In **lazy** conflict detection, the ownership will be acquired during the commit phase. In case of conflicts, choosing which transaction needs to be aborted is a responsibility of the *Contention Manager* (CM). There are many CMs proposed in the literature with different purposes. The most simple action that a CM can do is to abort the transaction that detected the conflict.

(a) Compact.

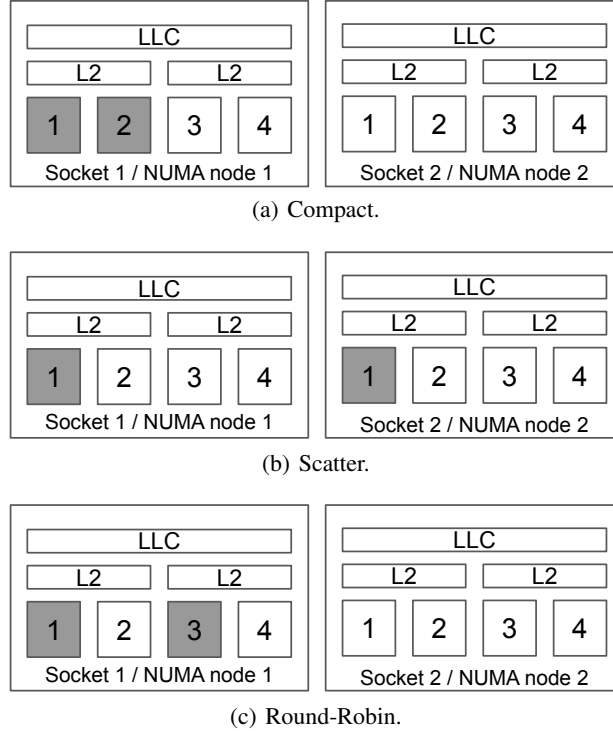(b) Scatter.

(c) Round-Robin.

Figure 1: Thread mapping strategies.

Regarding STM implementation, `TinySTM` [23] is one of the most used STM libraries and is considered a state-of-art STM implementation [11, 15]. `TinySTM` uses word granularity and has configurable version management and CMs. With respect to benchmarks, `STAMP` (*Stanford Transactional Applications for Multi-Processing*) [36] still is the most used STM benchmark suite [39]. This suite is composed of 8 applications with realistic characteristics, covering a wide range of transactional behavior, representing several application domains.

2.2 Thread Mapping

In thread mapping, threads are associated to cores, improving the cache usage and machine interconnections. The strategy utilized depends on the application and the underlying architecture. For instance, applications where a group of threads accesses the same shared data, these threads can be mapped to cores that are close to each other in the underlying architecture. If the application has a disjoint data access pattern, where each group of threads accesses different groups of data, mapping them on different cores processors can improve performance, since more cache will be available to each thread. The default scheduler used by Linux kernel, called *Completely Fair Scheduler* (CFS) [49] mainly focuses on load balancing. If the behavior of an application is known in advance and it does not change during execution, a static thread mapping can be used. For instance, Figure 1 shows different static thread mapping strategies with distinct objectives:

- **Compact** places threads on sibling cores that share all cache levels, thus potentially reducing the data access latency if neighboring threads communicate often (Figure 1(a)).

- **Scatter** distributes threads across different processors, avoiding cache sharing, thus, reducing memory contention (Figure 1(b)).

- **Round-Robin** is a mix between compact and scatter, where only the last level of cache (LLC) is shared (Figure 1(c)). It is worth noting that if the processor architecture employs lower cache levels as private to each core, for instance, the L2, then the resulting mapping will be exactly the same as Compact.

Regarding locality of the memory access, some techniques are used to keep track of the sharing behavior of the application in order to map threads to cores based on the memory access behavior. Thread mapping based on the

3

memory access behavior of applications is called sharing-aware[1] thread mapping [12]. There are many techniques in literature to perform an efficient sharing-aware thread mapping [19]. One of the main challenges of this area in shared memory architectures is to detect which threads are accessing each memory address. An affinity measure is necessary to be able to quantify the groups of threads that have more affinity. A communication or sharing matrix is the most common measure to determine the affinity between threads [4, 35]. Each cell in the matrix represents the amount of communication between pairs of threads. Figure 2 shows examples of communication matrices, where axes are thread IDs. In Figure 2(b), the matrix is represented graphically, where darker cells indicate more communication between
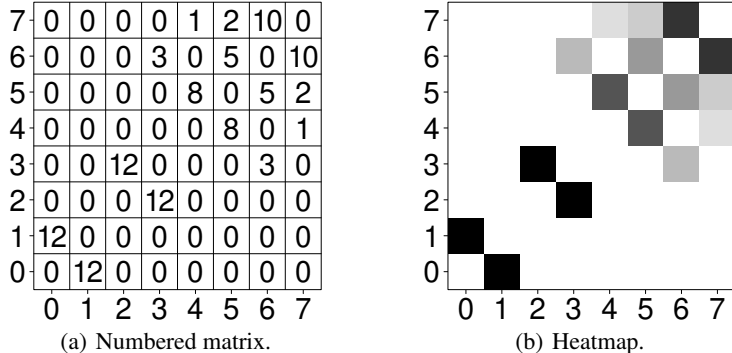

(a) Numbered matrix.   (b) Heatmap.

Figure 2: Examples of communication matrices.

pairs of threads.

The communication matrix can be used as an input to task-mapping algorithms. These algorithms use the hierarchical topology of the machine and the communication matrix to calculate an optimized mapping of threads to cores. Many tools are available to calculate a thread mapping based on a communication matrix, for instance, Scotch [41], TreeMatch [33], EagerMap [13], ChoiceMap [46], and TopoMatch [31].

2.3 Data Mapping

In data mapping, memory pages are associated to NUMA nodes, optimizing the usage of memory controllers. Similar to thread mapping, the strategy utilized depends on the application. The most common strategies are load balance or locality. In *locality*, memory pages are mapped to the same NUMA node where the core that most accesses them is located. The Linux kernel implements data mapping strategies in the form of memory allocation policies. The default policy is called *first-touch* [24] where memory is allocated in the NUMA node where the first access to the memory page is performed. Another data mapping policy available is *interleave*, which focuses on balance, allocating pages in a round-robin way on the NUMA nodes [34].

The default Linux kernel already has routines to improve memory locality of NUMA nodes. It keeps track of page faults, moving a page automatically to the node that most accessed it. This mechanism is called *NUMA balancing* [48].

3 Thread and Data Mapping in STM

This section surveys the related work on thread and data mapping in the context of software transactional memory. The works described in this section are summarized in Table 1. When describing the works in the following sections, we opted to group the works by authors since the majority of the authors proposed more than one approach to this subject.

3.1 Castro et al.

To the best of our knowledge, the first work to use transactional information to guide thread mapping was proposed by Castro et al. [9]. The motivation is that STM adds new challenges to thread mapping, such as different kinds of conflict detection and resolution. Hence, the best thread mapping depends on the STM configuration. To illustrate this scenario, the authors showed an experiment using all applications from the STAMP [36] benchmark using three different state-of-art STM libraries, TinySTM [23], SwissSTM [21] and TL2 [16] and the three thread mappings as

---

[1]This research field is also known as affinity-based [19] or topology-aware [32] mapping.

Table 1: Comparison of the proposed works to deal with thread and data mapping in STM applications.

Work	Year	Thread mapping	Data mapping	Mapping offline or online	Needs prior knowledge of the application?	Data used to define the mapping	Changes to the application
Castro et al. [9]	2011	✓		Offline	Yes	Abort ratio, LLC miss ratio, transaction time ratio and ML algorithm	No
Castro et al. [8]	2012	✓		Both	Yes	Abort ratio, LLC miss ratio, transaction time ratio and ML algorithm	No
Castro et al. [7]	2014	✓		Both	No	Abort ratio, LLC miss ratio, transaction time ratio and ML algorithm	No
Góes et al. [26]	2012	✓		Online	Yes	Sharing pattern (worklist)	Yes
Góes et al. [25]	2014	✓	✓	Both	Yes	Sharing pattern (worklist)	Yes
Chan et al. [10]	2015	✓		Online	No	Sharing pattern (based on aborts)	No
Zhou et al. [51]	2016	✓		Online	No	Throughput (between all mappings)	No
Zhou et al. [52]	2018	✓		Online	No	Throughput	No
Pasqualin et al. [38]	2020	✓		Offline	Yes	Sharing pattern (based on reads and writes)	No
Pasqualin et al. [37]	2020	✓		Online	No	Sharing pattern (based on reads and writes), abort ratio and number of accessed memory addresses	No
Pasqualin et al. [40]	2021		✓	Online	No	Sharing pattern (based on reads and writes)	No

shown in Figure 1. The best mapping varies according to the STM used. This is due to the different strategies used for each STM to deal with conflicts.

The authors proposed to feed a machine learning (ML) algorithm to discover the best thread mapping for each application. The ML algorithm used was the *Iterative Dichotomiser 3* (ID3) [42] which is based on decision trees. For each kind of conflict resolution and contention management strategy, they decided to collect the abort ratio (aborts divided by the total of transactions), LLC miss ratio (cache misses divided by the total accesses) and, transaction time ratio (transaction time / total execution time). After that, a second step normalizes the data and removes duplicates. This processed information is used as an input to ID3 for the training step. The final result is a decision tree called *predictor*, that is able to predict the best thread mapping according to the input data. It is worth noting that the machine used to collect the information influences the predictor, i.e., it is machine-dependent. Analyzing the generated predictor, they concluded that the abort ratio and LLC miss ratio are strongly related to the mapping predicted. For instance, when the abort ratio is low, the preferred mapping is scatter, as the application accesses few shared data.

Finally, the generated predictor was implemented inside the TinySTM [23] library and, using the STAMP [36] benchmark, it was possible to improve the execution time of applications, even when compared to an oracle that gives the best case.

The main drawback of the proposed approach is that thread mapping is chosen by the predictor only once during the execution of the application. However, for applications whose behavior changes during runtime, a more dynamic approach to adapt thread mapping for each new phase might be necessary. Hence, Castro et al. [8] extended their previous work, creating an approach that changes the thread mapping dynamically during runtime. The basic approach is the same as the previous work: use ID3 to generate a predictor being able to predict the best thread mapping according to the input parameters. A profiling interval and an interval between profiles are used to collect the necessary information for the predictor. Both intervals are based on a predefined amount of commits. The interval begins short and doubles each time that the thread mapping is not changed. When the mapping changes, the interval is reset to

the initial value. It is worth noting that, to avoid high overhead, only one thread is used to be the profiler thread. Since, according to the authors, the `STAMP` benchmarks do not present dynamic behavior during runtime, they used the synthetic `EigenBench` [30] benchmark to test the proposed mechanism. Using this benchmark, they created 56 synthetic applications, each one presenting 3 distinct execution phases. Overall the proposed dynamic mechanism switched correctly to the best thread mapping on the majority of applications.

In [7], the authors tested a new ML algorithm, called `Apriori` [1]. Another contribution is a new dynamic thread mapping technique where it is not necessary to have any prior knowledge of the applications. Different from the ML approaches used in their previous work, the proposed technique does not need preliminary runs of applications to collect information to guide mapping. As described in [9], the abort ratio has a strong relationship with the best thread mapping. Hence, the idea is to use this single metric to choose an optimal thread mapping dynamically while the application is running. The application starts with the default mapping chosen by the default Linux scheduler. After that, a profiler collects information about the abort ratio during n committed transactions. When the profiling phase ends, according to the abort ratio, a thread mapping is chosen between compact, scatter, and round-robin. Then, this new thread mapping is active until the next profiling phase ends. This strategy was called `Conflict`. Beyond that, another strategy called `Test-and-map` was proposed. In this strategy, the profiling phase is divided into three parts. On each part one thread mapping is activated and the execution time is measured. When the profile phase ends, the mapping that achieved the shortest execution time is selected to be active until the next profile time ends.

All proposed strategies, i.e, `ID3` and `Apriori` that are based on ML and `Conflict` and `Test-and-map` that are based on a single metric, were tested using the `STAMP` [36] and the 56 synthetic applications constructed with `EigenBench` [30]. All strategies showed performance gains when compared to the default Linux scheduler. As expected, the ML approaches showed higher gains, since they use more information to decide the mapping.

3.2 Góes et al.

Goés et al. [26] used the concept of skeleton or pattern-based programming, where a parallel program has a well-defined communication and computation pattern during execution. More specifically they focused on a *worklist* pattern, where each *work unit* to be processed is dynamically managed by a *worklist*, i.e, a collection of work unit instances. This pattern was chosen by observing that the majority of the `STAMP` [36] applications have this pattern.

They proposed the `OpenSkel` framework in the form of a runtime system library to be used for transactional worklist applications. This library provides an API (*Application Programming Interface*) to deal with transactional worklists, applying performance optimizations and autotuning during runtime. These tunings are possible due to pattern-based programming used, as applications present a well-defined computation pattern. Thus, it is possible to know in advance what an application will do next. The main primitives provided by `OpenSkel` are to allocate, run and free a worklist. It is worth noting that a transactional application needs to be designed or modified to use the proposed `OpenSkel` framework. The worklist is represented internally by the `OpenSkel` using a stack data structure, allowing to improve data locality if consecutive work units access common shared data. As the STM does not manage thread scheduling and mapping, the OS is responsible for this task. Since the default Linux scheduler tries to perform load balancing, and migrate threads during runtime, the authors opted for a fixed *scatter* thread mapping strategy statically when the application starts. Hence, since the thread mapping was set manually, the OS will not be allowed to migrate threads during runtime, allowing it to have more predictable performance. Besides, according to the authors, scatter is the mapping more similar to the default Linux strategy of load balance.

The `OpenSkel` proposes optimizations to boost the performance. One specific technique is the use of helper threads (HTs) that do not change the state of the application. Their main objective is to prefetch data, bringing them to the caches. HTs can be used if there are idle cores in the system. More specifically, idle cores that share cache levels with cores running transactions.

For the tests, they chose 5 five applications from `STAMP` [36] benchmark that present the worklist pattern: intruder, kmeans, labyrinth, vacation, and yada. These applications were modified to use the `OpenSkel` framework. The underlying STM system used to manage transactional operations was `TinySTM` [23]. Results indicate that, overall, the proposed framework improves the performance of applications, including when compared to an oracle approach.

In [25] the `OpenSkel` framework was extended, adding a new module responsible to deal with memory affinity. The main focus is to improve the performance of STM applications on NUMA architectures, by using page allocation policies. The HTs used for prefetching data together with page allocation policies were integrated into a mechanism called `SkelAff`, a submodule of `OpenSkel`, responsible to improve the memory affinity. Since the framework knows the next work unit that needs to be executed, `SkelAff` can trigger an HT to prefetch data that will be required soon in advance. Besides, it can allocate memory pages close to a specific thread if the mechanism knows in advance that there are work-units in the queue to be executed that are memory-related with work-units being executed. The

memory page policies can be *bind* or *cyclic*. The *bind* allocation reduces the access latency by binding data used by a thread on a single NUMA node. On the other hand, *cyclic* distributes memory pages using a round-robin strategy, focusing on load balance. By using information about the work-units, `OpenSkel` chooses the best memory page allocation policy and triggers HTs when necessary.

For the experiments, the focus was to verify the efficiency of the proposed `SkelAff` mechanism. They chose four applications from `STAMP` [36] benchmark that present the worklist pattern: intruder, kmeans, vacation, and yada (labyrinth was not used in this new experiments) and two distinct NUMA machines. Results indicate that the proposed framework improves the performance of applications and that a *cyclic* page allocations deliver more performance gains.

3.3   Chan et al.

In Chan et al. [10] two mechanisms were proposed. The first one is a dynamic concurrency control mechanism, used to limit the maximum number of concurrent threads executing transactions. The idea is that an excessive number of threads can hurt performance in a high contention environment, mainly due to a higher number of aborts. Thus, it is better to have a lower number of threads executing transactions than a high number of aborts. The second proposed mechanism keeps track of transactional conflicts to detect which threads are accessing the same shared data. Hence, based on this information, an *affinity-aware thread migration* is proposed. The main objective is to improve the cache usage of shared data managed by the STM runtime, consequently improving the execution time through enhanced cache locality. Also, they focus on multiprocessor systems. Since the concurrency control mechanism is out of the scope of this paper, we will focus on the affinity-aware thread migration approach.

When a conflict is detected, before aborting, the mechanism verifies the identifier (id) of the conflicting thread. Hence, this relationship is updated in a matrix. The value stored in each matrix position is derived from the formula of *contention intensity*, proposed initially by Yoo and Lee [50]. The derived formula was called *pairwise contention intensity* (PCI). The value stored in the i, j position of the matrix means how likely thread i is to be obstructed by thread j. The matrix can be seen as a directed adjacency matrix, where nodes represent threads and edges weights are the PCI between a pair of threads. Using a graph partitioning algorithm, the objective is to reduce the sum of edges that span across different cores processors. To avoid overhead, only one pair of threads is migrated between processors each time that the mechanism is triggered.

During a commit, the committing transaction, for instance, transaction i, updates its conflict intensity with all other threads (transactions), i.e, the PCI must be computed for all other running threads and updated in the matrix. The PCI used during the commit phase is: $C_{ij} \leftarrow C_{ij} \times 0.9$. On the other hand, when a transaction aborts, the PCI against the conflicting transaction is updated in the matrix, using the formula $C_{ij} \leftarrow C_{ij} \times 0.9 + (1 - 0.9)$. Periodically, a daemon thread applies a graph partition algorithm in the matrix to choose the pair of threads that should be migrated. It is worth noting that the algorithm to calculate the migration policy was designed to work on dual-processors systems only.

Both mechanisms, concurrency control and affinity-aware thread migration, were implemented inside the `TinySTM` [23] library. Threads were pinned to cores using the `pthread_setaffinity_np` function. For the experiments, authors used the `STAMP` [36] benchmarks. Also, the proposed mechanism was compared against the transactional schedulers ATS [50] and Shrink [22]. In `STAMP`, the affinity-aware thread migration does not improve performance. According to the author, `STAMP` does not present any specific thread correlation. However, to demonstrate the efficiency of the proposed technique, the authors created a synthetic dual Red-Black tree application, where it was possible to control the thread correlation, being possible to improve the performance with the proposed thread migration mechanism.

3.4   Zhou et al.

Similar to Chan et al. (Sect. 3.3), Zhou et al. [51] also proposed a concurrency control mechanism to dynamically adjust the maximum number of threads allowed to run transactions concurrently. As the number of active threads changes, the thread mapping is changed as well. Two key pieces of information are used by the authors in their proposed mechanism: the commit ratio (CR) calculated by the total of commits divided by the total of commits plus aborts and; throughput, i.e., the total of commits in a unit of time. The motivation to use both metrics are twofold: (1) throughput is influenced by the number of threads, i.e., a low throughput can be a consequence of a low number of threads running and;(2) CR is influenced by throughput, i.e, a low CR can present a high throughput if there is a high number of threads running transactions.

The mechanism starts with a probabilistic model calculating the predicted optimum number of threads allowed to run concurrently. During a profiling phase, the throughput is measured and compared to the previous one collected before changing the number of threads. If the current throughput is lower than the previous, the number of threads is switched back to the previous value. A second step is related to thread mapping. After deciding the thread number, if the CR changed (in a predefined range) and the thread number is less than half of the maximum core number, a second profile phase begins to decide the thread mapping. According to the authors, thread mapping has little impact on the performance when the number of cores is closer to the maximum cores of the machine, where all mappings perform similarly. For this reason, thread mapping is only calculated when the number of concurrent threads running is less than half of the cores of the machine. During the thread mapping profiling phase, four mappings are tested: Linux, scatter, compact, and Round-robin. In the end, the thread mapping that presented the higher throughput is set. After that, the mechanism restarts. An important aspect is that the information used by the profile is collected on all running threads, instead of, as in other works, based on a single thread. Hence, synchronization costs are added to gather the required profile information.

For the experiments, 6 applications from `STAMP` [36] (bayes and kmeans were not used) and one synthetic from `EigenBench` [30] were used. According to the authors, `STAMP` applications present similar behavior between transactions, i.e., overall, each transaction executes the same transactional code. Hence, they are not suitable to evaluate online thread mapping strategies. For this specific purpose, `EigenBench` was used. They conclude that due to the overhead, the proposed mechanism can benefit only applications that present dynamic behavior variation during execution. Finally, the authors discovered that thread mapping is influenced by the CR. In most cases, when the CR was low, compact was chosen, whereas when the CR was high, scatter was preferred.

In [52], together with the probabilistic model, the authors also proposed a *simple model* that searches for the near-optimum thread number. In this model, the thread number is incremented or decremented by one, according to the measured throughput in each profile phase. Regarding thread mapping, adjustments were made in order to reduce overhead. In their previous work, all mappings were tested during the profile phase, measuring the throughput and setting the mapping at the end. In the new approach, the thread mapping profile starts with Linux and then moves to the round-robin mapping. The compact mapping is tested only if round-robin throughput was better than Linux. Finally, scatter is tested only if the throughput of compact was worse than round-robin. When scatter is the best mapping, the overhead will be the same as the previous approach since all mappings will be tested.

The experiments and conclusions were similar to their previous paper. Nevertheless, they also concluded that the proposed mechanism does not scale on NUMA architectures, since the proposed mechanism does not take into consideration memory location and has excessive thread migrations to test the best thread mapping.

3.5   Pasqualin et al.

Pasqualin et al. [38] used sharing-aware thread mapping (Sect. 2.2) in the context of STM, to map threads to cores considering their memory access behavior. Contrary to previous sharing-aware mapping proposals that rely on memory traces of the entire application, they argue that tracking only STM operations to determine the sharing behavior has lower overhead and better accuracy because only memory accesses that are in fact shared between threads are traced. The main intuition used is that STM has precise information about shared variables and has native access to all information needed to characterize the sharing behavior, i.e., accessed memory addresses and the intensity with which each thread accesses them.

One of the main challenges of sharing-aware thread mapping in shared memory architectures is to detect which threads are accessing each memory address. However, word-based STM has native access to these pieces of information since they need to keep track and save versions of shared reads and writes to data. The proposed mechanism is triggered on each transactional read or write operation. An auxiliary hash table maps memory addresses to the last 2 threads that accessed them. When at least 2 distinct threads access the same memory address, a communication event between them is updated in a communication or shared matrix. The communication matrix generated by the proposed mechanism is sent to `EagerMap` [13], a task mapping algorithm for sharing-aware mapping, to generate the optimized thread mapping.

The proposed mechanism to collect the communication matrix was implemented inside the STM library `TinySTM` [23]. It is necessary to run each application individually in order to extract the communication matrix. After that, `EagerMap` generates the optimized thread to core mapping, and the application is re-executed pinning threads to cores using the function `pthread_setaffinity_np`.

For the experiments, the authors chose `STAMP` [36] and two synthetic applications, *Hashmap* and *Red-black tree*. The comparison of the proposed mechanism was made against compact, scatter, and round-robin mappings. Also, applications were executed in a NUMA machine using Opteron processors. Not all tested applications were suitable

for the proposed sharing-aware mapping, including reducing the number of aborts. Finally, it is demonstrated that the overhead to collect the communication matrix is much lower than using other tools that trace all memory addresses, such as numalize [20].

In [37], the proposed mechanism was modified to collect the communication matrix and perform the thread mapping during the execution of an application. The new mechanism was called STMap. To reduce the overhead of the mechanism, the authors sampled the accessed memory addresses to fill the communication matrix. They have made experiments to determine the best *sampling interval* (SI) to have a low overhead but a high accuracy of the collected communication matrix. Thus, a SI of 100 was selected. Besides, it was necessary to define the *mapping interval* (MI), i.e, when the thread mapping should be calculated and performed. Based on experiments and observations, they chose a MI of between 50,000 and 100,000 addresses accessed (total, not only sampled) by the main thread. Regarding calculating the thread mapping, in STMap, EagerMap [13] was replaced by TopoMatch [31].

Since STMap works during runtime, the authors tried to create a heuristic to identify the applications that are not suitable to sharing-aware thread mapping, thus, disabling STMap for these applications, i.e., not applying the new thread mapping during the execution. The heuristic is based on the number of distinct memory addresses accessed by the application and the abort and commit ratios. The new thread mapping should be calculated if; when the MI is triggered the application accessed less than 10,000 distinct memory addresses. If more than 10,000 addresses were accessed, the heuristic verifies if the abort ratio is greater than the commit ratio. If true, the thread mapping should be calculated as well.

The methodology of the experiments was similar to the previous work. Nevertheless, they included a Xeon machine in the experiments and the previous static mechanism to the comparison. The proposed heuristic to identify which applications are suitable for sharing-aware thread mapping worked correctly for the majority of applications, improving execution time when compared to the static mechanism. Hence, taking into consideration the improved execution time over all benchmarks, STMap was the best mapping.

Finally, in [40] STMap was extended to include sharing-aware data mapping. However, the authors conclude that, contrary to thread mapping where only taking into consideration STM access is sufficient to have a global vision of the sharing behavior, for data mapping this observation is not enough. For data mapping, it is necessary to have a global vision of memory pages accessed to be able to perform an optimized data mapping, not only the memory pages accessed by the sharing data protected by STM runtime.

3.6   Discussion

We will start by discussing data mapping. As it is possible to observe in Table 1, just two works attempted to deal with data mapping in STM. The first one proposed by Góes et al. [25] provided a mechanism to choose if the data allocated inside the STM runtime should be allocated in the same NUMA node as the thread that is needing it or prioritizing a memory balance. Hence, it is based on a static (offline) mapping. However, as discussed by Pasqualin et al. [40] it could be infeasible to perform an efficient online data mapping by only having accesses to the memory pages accessed by the STM runtime since it represents only a fraction of the entire memory accessed by the application. As the opposite, for thread mapping, the information provided by the STM runtime is exactly the one needed to perform an efficient thread mapping. One of the main objectives of thread mapping is to improve cache usage by mapping cores that access the same shared data in a way that they can share different cache levels.

Regarding thread mapping, the information about aborts, such as amount and ratio compared to commits, can be a useful metric to define thread mapping. The intuition used is that if the aborts are high, then the application is accessing a great quantity of share data. Hence, it can be interesting to use a mapping strategy that prioritizes locality. In fact, many works used abort information to decide the thread mapping.

There are works that instead of relying solely on the number of aborts to identify a global shared pattern, try to identify which specific threads are accessing the same shared data. Chan et al. [10] tried to identify the relationship between threads when they abort. However, their mechanism has a high overhead since it is triggered on each abort and the affinity measured should be recalculated for all threads each time. Pasqualin et al. [37] used a similar idea, but tried to improve the accuracy of the mechanism by tracking transactional reads and writes, instead of only aborts. The mechanism has a lower overhead since the affinity measure is a counter updated once for each access. Also, transactional reads and writes are only sampled.

A different strategy was proposed by Zhou et al. [51, 52] to identify the best thread mapping. When the number of concurrent threads executing transactions is changed, a new thread mapping is set to match with the new number of threads. Their mechanism measures the throughput of each thread mapping and choses the one that presents higher throughput at the end of the test phase. Although the proposed mechanism works correctly, it incurs on high overhead

because of the thread migrations triggered to test all mappings. As pointed out by the authors, this mechanism does not scale well on NUMA machines due to the excessive thread migrations.

Some mechanisms need previous knowledge of the applications. Using this knowledge it is possible to perform an offline thread mapping at the beginning of the execution of the STM application. Although the main disadvantage of these offline mechanisms is the need for a preliminary run to gather the needed information about the application, during the execution of the application they present low overhead and can be more accurate and efficient when compared to online mechanisms [2]. Another related disadvantage is that the offline mechanism can be inefficient if the application changes the sharing behavior during the execution. As opposite, the online mechanism does not need a preliminary run to gather information about applications and can adapt if the sharing behavior suddenly changes during runtime. However, depending on the chosen strategy, the online mechanism can generate an infeasible overhead to be used during runtime.

Finally, one important characteristic of the proposed mechanisms is the need to change the STM application to be able to use it. Only the works proposed by Góes et al. [26, 25] have this need, i.e., the STM application needs to be created or modified to be able to benefit from the proposed optimizations. All others proposed mechanisms only change the STM runtime. Hence, not being necessary to change the STM applications.

4   Limitations of existing proposals and future research directions

In this section we will discuss some limitations and opportunities of research regarding thread and data mapping in Transactional Memory.

The first observation is that no works are focusing on HTM or hybrid TM systems, only STM. Also, all works used the `TinySTM` [23] library as a base TM to implement the proposed mechanisms. Hence, it is not clear how much thread and data mapping can be helpful for other systems or STM implementations. The same observation can be made for distributed memory systems. All works presented in this survey are focused on STM running on shared memory systems.

Excepting the work of Pasqualin et al. [38] there are no works that verified the relationship between aborts and thread mapping, i.e, if they influence each other. Besides, the discussion presented in [38] only pointed out that there can be a relationship between them. However, it is necessarily an in-depth investigation, analyzing each application individually to make strong conclusions about the topic. This leads us to another research opportunity: how thread and data mapping behave when used with techniques that focus on reducing the number of aborts, such as transactional schedulers [28, 14]. This relationship was also not studied in any related work. Similarly, there are no works that measured how much energy was saved by the optimized thread and data mapping using only the information provided by the STM runtime.

Not specific about thread and data mapping, but an issue that can influence experiments testing proposed mechanisms, is the lack of new and updated benchmarks for TM systems. By far the most used benchmark suite for TM is the `STAMP` [36]. However, as pointed out in many works, `STAMP` is not ideal for testing online thread and data mapping since all threads present similar behavior, executing the same transactional code, and the sharing behavior stays the same during execution time [39, 8, 7, 25, 51, 52]. It would be interesting changing, or adding new applications to `STAMP` that present distinct sharing behaviors during execution or that have groups of threads doing different transactional jobs.

For online thread and data mapping mechanisms, it is necessary to extract, with low overhead, information about the application during runtime. For this purpose, as far as we know, only two works were proposed for this specific purpose [6, 44].

Finally, regarding machine learning (ML) techniques, only Castro et al. [9, 8, 7] explored this approach. However, their mechanism relies on offline techniques, needing a previous execution of the application to collect the data for the training phase. Recent works on thread and data mapping for general applications used successfully ML techniques in an online context [43]. Hence, this online ML approach could be revisited in the context of transactional memory.

5   Conclusions

Due to complex memory hierarchies and different latencies in memory accesses on multicore processors, locality of memory accesses is an essential issue for parallel application performance. Thread and data mapping are techniques to improve locality based on using information about the memory access behavior of a parallel application to optimize

the location of threads and memory pages on NUMA systems. STM is an abstraction for thread synchronization that presents new challenges and opportunities for different types of thread and data mapping.

This paper surveyed proposals that solely use the information provided by an STM runtime to guide thread and data mapping. We found that an STM runtime can provide sufficient information to perform an efficient and effective thread mapping. However, for data mapping, results indicate that only taking transactional information into consideration is not enough to perform an efficient data mapping, since the STM runtime can only discover a fraction of the entire memory accessed by an application. Finally, we presented research opportunities and directions where this research area could be expanded. Some of research opportunities for thread and data mapping in the TM domain, for instance, are on HTM, integration to transactional schedulers, energy efficiency, machine learning, among others.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[2] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, and Suraj Kumar. Are static schedules so bad? a case study on cholesky factorization. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1021–1030, Washington, DC, USA, may 2016. IEEE CS.

[3] Gary Anthes. Researchers simplify parallel programming. *Commun. ACM*, 57(11):13–15, October 2014.

[4] Cyril Bordage and Emmanuel Jeannot. Process affinity, metrics and impact on performance: An empirical study. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '18, pages 523–532, Piscataway, NJ, USA, 2018. IEEE Press.

[5] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. How to implement any concurrent data structure for modern servers. *SIGOPS Oper. Syst. Rev.*, 51(1):24–32, September 2017.

[6] Márcio Castro, Kiril Georgiev, Vania Marangozova-Martin, Jean-François Méhaut, Luiz Gustavo Fernandes, and Miguel Santana. Analysis and tracing of applications based on software transactional memory on multicore architectures. In *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 199–206, Washington, DC, USA, feb 2011. IEEE.

[7] Márcio Castro, Luís Fabrício W. Góes, and Jean-François Méhaut. Adaptive thread mapping strategies for transactional memory applications. *Journal of Parallel and Distributed Computing*, 74(9):2845 – 2859, 2014.

[8] Márcio Castro, Luís Fabrício Wanderley Góes, Luiz Gustavo Fernandes, and Jean-François Méhaut. Dynamic thread mapping based on machine learning for transactional memory applications. In Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, pages 465–476, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[9] Márcio Castro, Luís Fabrício Wanderley Góes, Christiane Pousa Ribeiro, Murray Cole, Marcelo Cintra, and Jean-François Méhaut. A machine learning-based approach for thread mapping on transactional memory applications. In *2011 18th International Conference on High Performance Computing*, pages 1–10, Washington, DC, USA, 2011. IEEE CS.

[10] Kinson Chan, King Tin Lam, and Cho-Li Wang. Cache affinity optimization techniques for scaling software transactional memory systems on multi-CMP architectures. In *2015 14th International Symposium on Parallel and Distributed Computing*, pages 56–65, Washington, DC, USA, June 2015. IEEE Computer Society.

[11] Daming D. Chen, Phillip B. Gibbons, and Todd C. Mowry. TardisTM: Incremental repair for transactional memory. In *Proceedings of the Eleventh International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '20, pages 1–10, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Eduardo H. M. Cruz, Matthias Diener, and Philippe O. A. Navaux. *Thread and Data Mapping for Multicore Systems*. Springer Publishing Company, Cham, Switzerland, 2018.

[13] Eduardo H. M. Cruz, Matthias Diener, Laércio L. Pilla, and Philippe O. A. Navaux. EagerMap: A task mapping algorithm to improve communication and load balancing in clusters of multicore systems. *ACM Trans. Parallel Comput.*, 5(4), March 2019.

[14] Pierangelo Di Sanzo. Analysis, classification and comparison of scheduling techniques for software transactional memories. *IEEE Trans. Parallel Distrib. Syst.*, 28(12):3356–3373, dec 2017.

[15] Pierangelo Di Sanzo, Alessandro Pellegrini, Marco Sannicandro, Bruno Ciciani, and Francesco Quaglia. Adaptive model-based scheduling in software transactional memory. *IEEE Transactions on Computers*, 69(5):621–632, May 2020.

[16] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[17] Diego Didona, Nuno Diegues, Anne-Marie Kermarrec, Rachid Guerraoui, Ricardo Neves, and Paolo Romano. ProteusTM: Abstraction meets performance in transactional memory. *SIGARCH Comput. Archit. News*, 44(2):757–771, March 2016.

[18] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, PACT '14, pages 3–14, New York, NY, USA, August 2014. Association for Computing Machinery.

[19] Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Philippe O. A. Navaux, and Israel Koren. Affinity-based thread and data mapping in shared memory systems. *ACM Comput. Surv.*, 49(4):64:1–64:38, December 2016.

[20] Matthias Diener, Eduardo H.M. Cruz, Laércio L. Pilla, Fabrice Dupros, and Philippe O.A. Navaux. Characterizing communication and page usage of parallel applications for thread and data mapping. *Performance Evaluation*, 88-89:18–36, jun 2015.

[21] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 155–165, New York, NY, USA, 2009. ACM.

[22] Aleksandar Dragojević, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 7–16, New York, NY, USA, 2009. ACM.

[23] Pascal Felber, Christof Fetzer, Torvald Riegel, and Patrick Marlier. Time-based software transactional memory. *IEEE Transactions on Parallel & Distributed Systems*, 21:1793–1807, 2010.

[24] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management on modern NUMA systems. *Commun. ACM*, 58(12):59–66, November 2015.

[25] Luís Fabrício Góes, Christiane Pousa Ribeiro, Márcio Castro, Jean-François Méhaut, Murray Cole, and Marcelo Cintra. Automatic skeleton-driven memory affinity for transactional worklist applications. *Int. J. Parallel Program.*, 42(2):365–382, April 2014.

[26] Luís Fabrício Wanderley Góes, Nikolas Ioannou, Polychronis Xekalakis, Murray Cole, and Marcelo Cintra. Autotuning skeleton-driven optimizations for transactional worklist applications. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2205–2218, 2012.

[27] Håkan Grahn. Transactional memory. *J. Parallel Distrib. Comput.*, 70(10):993–1008, October 2010.

[28] Danny Hendler and Adi Suissa-Peleg. Scheduling-based contention management techniques for transactional memory. In Rachid Guerraoui and Paolo Romano, editors, *Transactional Memory. Foundations, Algorithms, Tools, and Applications: COST Action Euro-TM IC1001*, pages 213–227. Springer International Publishing, Cham, Switzerland, 2015.

[29] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The Art of Multiprocessor Programming*. Elsevier, Cambridge, MA 02139, USA, second edition, 2020.

[30] Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, IISWC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[31] Emmanuel Jeannot. TopoMatch: Process mapping algorithms and tools for general topologies. https://gitlab.inria.fr/ejeannot/topomatch, 2021. Accessed 5 June 2021.

[32] Emmanuel Jeannot, Esteban Meneses, Guillaume Mercier, François Tessier, and Gengbin Zheng. Communication and topology-aware load balancing in Charm++ with TreeMatch. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, Washington, DC, USA, 2013. IEEE Computer Society.

[33] Emmanuel Jeannot, Guillaume Mercier, and François Tessier. Process placement in multicore clusters: Algorithmic issues and practical techniques. *IEEE Trans. Parallel Distrib. Syst.*, 25(4):993–1002, 2014.

[34] Christoph Lameter. NUMA (Non-Uniform Memory Access): An overview. *Queue*, 11(7):40:40–40:51, July 2013.

[35] Arya Mazaheri, Felix Wolf, and Ali Jannesari. Unveiling thread communication bottlenecks using hardware-independent metrics. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 1–10, New York, NY, USA, 2018. ACM.

[36] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46, Washington, DC, USA, Sept 2008. IEEE Computer Society.

[37] Douglas P. Pasqualin, Matthias Diener, André R. Du Bois, and Maurício L. Pilla. Online sharing-aware thread mapping in software transactional memory. In *2020 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 35–42, Washington, DC, USA, September 2020. IEEE CS.

[38] Douglas P. Pasqualin, Matthias Diener, André R. Du Bois, and Maurício L. Pilla. Thread affinity in software transactional memory. In *19th Int. Symposium on Parallel and Distrib. Comput. (ISPDC)*, pages 180–187, Washington, DC, USA, July 2020. IEEE CS.

[39] Douglas P. Pasqualin, Matthias Diener, André R. Du Bois, and Maurício L. Pilla. Characterizing the sharing behavior of applications using software transactional memory. In Felix Wolf and Wanling Gao, editors, *Benchmarking, Measuring, and Optimizing (Bench 2020)*, volume 12614 of *Lecture Notes in Computer Science*, pages 3–21, Cham, 2021. Springer International Publishing.

[40] Douglas P. Pasqualin, Matthias Diener, André R. Du Bois, and Maurício L. Pilla. Sharing-aware data mapping in software transactional memory. In Alex Orailoglu, Matthias Jung, and Marc Reichenbach, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 13227 of *Lecture Notes in Computer Science*, pages 481–492, Cham, 2022. Springer International Publishing.

[41] François Pellegrini. Static mapping by dual recursive bipartitioning of process architecture graphs. In *Proceedings of IEEE Scalable High Performance Computing Conference*, pages 486–493, Washington, DC, USA, 1994. IEEE Computer Society.

[42] John Ross Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[43] Isaac Sánchez Barrera, David Black-Schaffer, Marc Casas, Miquel Moretó, Anastasiia Stupnikova, and Mihail Popov. Modeling and optimizing NUMA effects and prefetching with machine learning. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, pages 1–13, New York, NY, USA, 2020. Association for Computing Machinery.

[44] Martin Schindewolf and Wolfgang Karl. Capturing transactional memory application's behavior – the prerequisite for performance analysis. In Victor Pankratius and Michael Philippsen, editors, *Multicore Software Engineering, Performance, and Tools*, volume 7303 of *Lecture Notes in Computer Science*, pages 30–41, Berlin Heidelberg, 2012. Springer-Verlag.

[45] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[46] Pirah Noor Soomro, Muhammad Aditya Sasongko, and Didem Unat. BindMe: A thread binding library with advanced mapping algorithms. *Concurrency and Computation: Practice and Experience*, 30(21):e4692, June 2018.

[47] John A. Trono. Transactions: They're not just for banking any more. *J. Comput. Sci. Coll.*, 30(5):160–166, May 2015.

[48] Rik van Riel and Shen Feng. Documentation for /proc/sys/kernel/. https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#numa-balancing, 2021. Accessed 5 June 2021.

[49] Chee Siang Wong, Ian Tan, Rosalind Deena Kumari, and Fun Wey. Towards achieving fairness in the Linux scheduler. *SIGOPS Oper. Syst. Rev.*, 42(5):34–43, July 2008.

[50] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 169–178, New York, NY, USA, 2008. ACM.

[51] Naweiluo Zhou, Gwenaël Delaval, Bogdan Robu, Éric Rutten, and Jean-François Méhaut. Autonomic parallelism and thread mapping control on software transactional memory. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 189–198, Washington, DC, USA, 2016. IEEE CS.

[52] Naweiluo Zhou, Gwenaël Delaval, Bogdan Robu, Éric Rutten, and Jean-François Méhaut. An autonomic-computing approach on mapping threads to multi-cores for software transactional memory. *Concurrency and Computation: Practice and Experience*, 30(18):e4506, may 2018.