Online Sharing-Aware Thread Mapping 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 alternative abstraction to synchronize processes in parallel programming. One advantage is simplicity since it is possible to replace the use of explicit locks with atomic blocks. Regarding STM performance, many studies already have been made focusing on reducing the number of aborts. However, in current multicore architectures with complex memory hierarchies, it is also important to consider where the memory of a program is allocated and how it is accessed. This paper proposes the use of a technique called sharing-aware mapping, which maps threads to cores of an application based on their memory access behavior, to achieve better performance in STM systems. We introduce STMap, an online, low overhead mechanism to detect the sharing behavior and perform the mapping directly inside the STM library, by tracking and analyzing how threads perform STM operations. In experiments with the STAMP benchmark suite and synthetic benchmarks, STMap shows performance gains of up to 77% on a Xeon system (17.5% on average) and 85% on an Opteron system (9.1% on average), compared to the Linux scheduler.**

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

I. Introduction

Since the introduction of multicore processors, the number of cores in a single chip has been growing every year, both on desktop and server processors. In order to better exploit the parallelism available in these modern architectures, software must be parallel and scalable [1]. The most used abstraction for process synchronization in parallel programming is mutual-exclusion locks. However, its semantics is not intuitive and its use error-prone, which can lead to a number of programming errors including deadlocks.

Transactional memory (TM) is an alternative abstraction for process synchronization in parallel programming [2]. The major advantage is its simplicity, since it is possible to replace the manual acquisition and release of locks with atomic blocks. Developers only need to delimit the block of code that they want to be executed as a transaction. All the complexity of ensuring a consistent execution, e.g., without deadlocks, is the responsibility of the TM runtime. Although there are TM implementations for hardware and software, in this paper, we focus on software TM (STM), where transaction consistency is guaranteed by a software library. An advantage of implementation in software is that it is more flexible and not dependent on hardware. Also, it does not have the same resource limitations as in hardware [1]. There are several proposals for increasing the performance of STM systems. However, the majority of them focuses on reducing the number of conflicts (transactional aborts). One technique is the use of a scheduler, acting proactively, using heuristics to prevent conflicts and to decide *when* and *where* a transaction should be executed [3].

In current multicore architectures with many sockets and cores, there are complex memory hierarchies and different latencies for memory access. Hence, a correct thread placement, that improves the use of memory controllers and data locality, is important to achieve good performance. A technique called *sharing-aware thread mapping* [4] aims to map threads to cores of an application considering their memory access behavior. Since STM is used to synchronize data accessed by multiple threads, an efficient thread mapping will help to make better usage of caches and memory controllers, hence improving the overall performance.

In this paper, we introduce *STMap*, an online, low overhead mechanism to perform sharing-aware thread mapping for STM applications. While a parallel application is running, STMap collects data about its inter-thread STM sharing behavior, uses that information to calculate an optimized mapping of threads to cores, and migrates the threads. We also add a heuristic to disable this mechanism if it determines that the sharing-aware mapping will not benefit the application. Since the STM runtime has precise information about shared data and threads that are accessing it, our mechanism can perform mapping accurately and with a low overhead.

Compared to previous proposals [5], [6], [7], [8] that rely on memory traces of the entire application, our mechanism has a lower overhead, because it only tracks memory accesses that are in fact shared between threads. Another important

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

advantage is that STMap does not need prior information about the behavior of the application, since detection and migration are performed based on information gathered solely during execution. It also requires no changes to existing STM applications. In experiments with the `STAMP` applications [9] and other synthetic benchmarks, STMap achieved performance gains of up to 77% on a Xeon machine (17.5% on average) and 85.8% on an Opteron machine (9.1% on average) compared to the Linux scheduler.

II. Background: Sharing-aware mapping in STM

A. Software Transactional Memory

Software transactional memory (STM) is an abstraction to synchronize accesses to shared resources. It replaces the use of locks with atomic blocks. Instead of manually acquiring and releasing locks, the developer only needs to put the critical section in an atomic block, which will be executed like a database transaction. If the transaction was executed without conflicts, a *commit* is done, i.e., all operations are made visible to other transactions. Otherwise, a transaction needs to *abort*, i.e., all operations inside the transaction must be discarded, and it restarts until a commit is possible.

B. Sharing-aware mapping

Data locality is an important factor in modern multicore and NUMA systems. One way to better exploit locality is to map threads according to their *memory access behavior* [4]. Mapping threads to cores while taking their memory access behavior into account can improve the cache usage and interconnection traffic.

One technique to map threads based on their memory access behavior is called *sharing-aware thread mapping* [4]. For instance, the Linux kernel already has routines to deal with thread mapping. However, the default Linux scheduler, called *Completely Fair Scheduler* (CFS), which mainly focuses on load balancing, does not take the memory access behavior into consideration when mapping threads to cores.

To determine the memory access behavior, it is necessary to know how threads share data [10]. The most common data structure used to represent the share information is a *communication matrix* [11]. A mapping algorithm uses information about the hardware hierarchy and the communication matrix to calculate an improved mapping of threads to cores.

III. STMap: An online mechanism for sharing-aware thread mapping in STM

A. Detecting communication in STM applications

To detect the memory access behavior of an application, it is necessary to know which addresses the application is accessing. In STM implementations, 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. Hence, it is not necessary to rely on external tools or add instrumentation overhead to get this information.

Detecting the communication behavior accurately incurs a high overhead [10]. Some tools such as `numalize` [5] were developed for this purpose. However, since STM runtimes have precise information about shared variables, it is possible to determine the communication behavior by tracking transactional reads and writes. Our proposed mechanism works as follows. When at least 2 distinct threads access the same address, a communication event between them is updated. The amount of communication between threads is stored in a communication matrix, where each matrix position represents the amount of communication between pairs of threads.

Our algorithm to detect the communication pattern of an application is shown in Algorithm 1. This algorithm is executed before each transactional read or write operation. To keep track of accessed addresses, a hash table is used where keys are memory addresses. Each position of the hash table contains a structure with the memory address and the last 2 threads that have accessed it. In case of hash conflicts, a linked list with all memory addresses with the same hash is kept. On line 1, the function `getAddressInfo` gets from the hash table the structure containing information about the address being accessed. The next step is to get how many accesses this address had before the current one (line 2). If

Algorithm 1 Detecting communication patterns.

Require:
addr: memory address being accessed
tid: thread ID of the thread that is accessing the address

▷ elem is a struct that stores the memory address and the last 2 threads that have accessed it

```
1:  elem ← getAddressInfo(addr)
2:  accesses ← getAccesses(elem)
3:  if (accesses = 0) then                    ▷ First access to the address
4:      elem.t1 ← tid
5:  end if
6:  if (accesses = 1) then                    ▷ One previous access
7:      if (elem.t1 ≠ tid) then
8:          update_comm_matrix(tid, elem.t1)
9:          elem.t2 ← elem.t1
10:         elem.t1 ← tId
11:     end if
12: end if
13: if (accesses = 2) then                    ▷ Two previous accesses
14:     if (elem.t1 ≠ tid) and (elem.t2 ≠ tid) then
15:         update_comm_matrix(tid, elem.t1)
16:         update_comm_matrix(tid, elem.t2)
17:         elem.t2 ← elem.t1
18:         elem.t1 ← tid
19:     else if (elem.t1 = tid) then
20:         update_comm_matrix(tid, elem.t2)
21:     else if (elem.t2 = tid) then
22:         update_comm_matrix(tid, elem.t1)
23:         elem.t2 ← elem.t1
24:         elem.t1 ← tid
25:     end if
26: end if
```

it is the first access (line 3), we only store the thread that is accessing it (line 4), i.e., no communication events have happened yet. The next case is if the address had one access before the current (line 6) and it was made by a different thread (line 7). If true, we have a communication event. In that case, we call the function `update_comm_matrix` (line 8) to update the communication matrix, increasing the amount of communication between the two threads. The matrix has an order equal to the maximum number of threads. Also, we update the threads that accessed the address (lines 9 and 10).

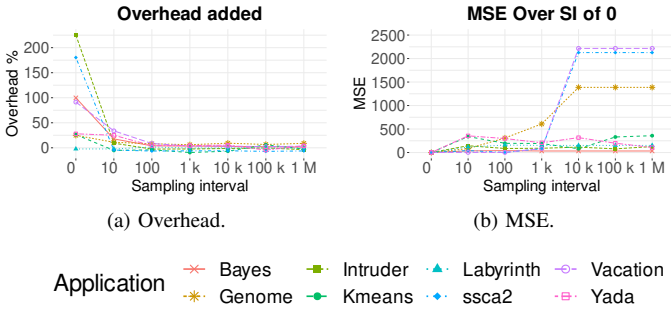
The final case is if the address had 2 previous accesses (line 13), then there are 3 subcases. The first one is if the current thread accessing the address is different from the 2 previous (line 14). In that case, we have a third distinct thread accessing the address, and we update the communication matrix between the 3 (lines 15 and 16). After that, the oldest access is removed (line 17). However, if the test of the line 14 was false, it means that the current thread accessing the address is the same from a previous one. In that case, we need to check which thread is the same (lines 19 and 21) and update the communication matrix correctly, removing old accesses, if necessary (line 23).

B. Reducing the overhead of online detection

Since we are interested in detecting communication and performing thread mapping online, keeping track of every accessed address would be infeasible, due to the high overhead added to the application. Hence we use the concept of *sampling*. The goal is to choose a *sampling interval* (SI) with a high accuracy but low overhead. To choose the SI, we ran an experiment using all eight benchmarks from the Stanford Transactional Applications for Multi-Processing (`STAMP`) [9]. `STAMP` is a collection of realistic workloads, covering a wide range of transactional execution cases.

We start using an SI of zero, i.e., all addresses are sampled in the communication matrix. The next SI is 10, i.e., only execute Algorithm 2 for every ten addresses accessed. To avoid contention, every thread has their own SI counter.

Fig. 1a shows the overhead results. Applications with many addresses accessed such as *Intruder* and *ssca2* have a high overhead even when using an SI of 10. We also studied how the accuracy of the collected communication matrices is affected by each SI. We used the mean squared error (MSE) [12] as a metric to compare the resulting matrices of each SI, comparing them to the SI of 0. Fig. 1b shows the results. Lower values are better. Analyzing the graphs, we chose an SI of **100**, where the applications presented the best trade-off between overhead and accuracy.



(a) Overhead. (b) MSE.

Fig. 1: Overhead and MSE when varying the SI.

Finally, it was necessary to choose the *mapping interval* (MI), i.e., when the new thread mapping should be calculated. A thread migration incurs an overhead, due to cache misses and other collateral effects. Also, since we have two NUMA machines (more details in Sect. IV-A), these effects could be even worse, due to messages of cache invalidation between nodes. Hence, our idea is to reduce the number of times that thread mapping is performed. Thus, we need to choose an MI as early as possible, taking into consideration the trade-off between accuracy and overhead. Our MI is based on the amount of accessed address (total, not only the sampled). Thus, we made a previous analysis of the applications and chose an MI between 50,000 and 100,000, according to the machine used (more details in Sect. IV-A). Hence, we calculate the new thread mapping when the application accessed MI addresses. Again, to avoid contention we decided to track the total of accessed addresses of only 1 thread.

C. Calculating the mapping

To calculate the thread mapping, the communication matrix created by the Algorithm 1 is used as input for `TopoMatch` mapping algorithm [13], which is an extension of `TreeMatch` [14]. Using the generated mapping, the threads are pinned to cores using the function `pthread_setaffinity_np`. Since we have NUMA machines for the experiments (more details in Sect. IV-A), `TopoMatch` has a desirable feature. When calculating the new thread mapping, it tries to minimize the communication costs between sockets/nodes. Hence, it prioritizes the placement of threads first inside the same socket.

Not all applications are suitable for sharing-aware mapping [5]. Thus, we use a heuristic which is measured before calculating a new thread mapping to verify if the application would benefit from using the proposed approach. This heuristic is shown in Equation 1.

$$enable\_m(da, ar, cr) = \begin{cases} true & \text{if da} < \text{da\_threshold} \\ otherwise \begin{cases} true & \text{if cr} < \text{ar} \\ false & \text{if cr} \geq \text{ar} \end{cases} \end{cases} \quad (1)$$

The heuristic is based on the *distinct address* (da) accessed by the application, the *abort ratio* (ar) and *commit ratio* (cr). Following the same intuition from MI, the ar and cr are determined only by thread 1. However, the da is global, because we need to calculate it only one time (from the hash table, Sect. III-A), inside the heuristic. The value of da_threshold was set to 10,000 based on previous analyses of the applications. Thus, if the application had accessed less than 10,000 distinct addresses, the new thread mapping should

be calculated. The intuition used is if there are a lot of distinct addresses, the application probably does not have a well-defined communication pattern between threads. However, if there are more than 10,000 distinct addresses, the commit and abort ratio of thread 1 are used for determining if the mechanism should be disabled. The intuition here is based on the works of Castro et al. [15] and Chan et al. [16], who determined that if the abort ratio is high, then the application is accessing too much shared data. Thus, putting them closer would increase the cache sharing between the shared data, which is one of the main objectives of sharing-aware thread mapping.

D. Final algorithm

In this Section, we present the final algorithm to detect and perform the thread mapping dynamically (Algorithm 2).

Algorithm 2 Triggering communication events and mapping.

Require:
addr: memory address being accessed
tx: struct that contains data about the current transaction
si: sample interval. Default 100
mi: map interval. Default 50,000 or 100,000

```
1:  tx.addr_sample ← tx.addr_sample + 1
2:  if (tx.addr_sample > si) then
3:      tx.addr_sample ← 0
4:      Execute algorithm 1
5:  end if
6:  if (tx.tid = 1) then
7:      tx.total_addr ← tx.total_addr + 1
8:  end if
9:  if (tx.tid = 1) and (tx.total_addr ≥ mi) then
10:     if (enable_m) then                    ▷ Equation 1
11:         Compute new thread mapping
12:     end if
13: end if
```

First, the thread private variable `addr_sample` (line 1) is incremented to verify if is time to sample the memory access. Then, on the line 2 we verify if the counter of the current thread is greater than the SI (Sect. III-B). If true, we zero the variable to be able to detect the next trigger time (line 3), and Algorithm 1 is executed.

The next part of the algorithm controls when to perform the new thread mapping. On line 6, we determine if the current thread is the one responsible to control the total of addresses accessed, i.e., thread one. If true, we increment the thread private variable `total_addr` (line 7). Thus, on line 9 the algorithm determines if it is time to trigger the new thread mapping, checking if the total of accesses is greater or equal to MI (Sect. III-B). If true, it is necessary to verify if the application will benefit from a new mapping. This verification is done through Equation 1 (line 10). Finally, if Equation 1 returns *true*, we compute the new thread mapping according to Sect. III-C.

E. Implementation

The proposed mechanism presented in this Section was implemented inside the `TinySTM` library [17], version 1.0.5. Algorithm 2 is executed 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-E), with the default configuration: *lazy* version management, *eager* conflict detection and contention manager *suicide*.

The applications used in the experiments were all eight benchmarks from the Stanford Transactional Applications for Multi-Processing (`STAMP`) [9] version 0.9.10, and two micro-benchmarks (HashMap and Red-Black tree) from [18]. To run the experiments, we used two NUMA machines with the follow characteristics:

- **Xeon**: 8 Intel Xeon E5-4650 processors and 488 GiB of RAM running Linux kernel 4.19.0-9. Each socket has 12 2-HT cores, totaling 96 cores. Each socket corresponds to a NUMA node (for a total of 8 NUMA nodes), and 12× 32 KB L1d, 12× 32 KB L1i, 12× 256 KB L2 and 30 MB L3 cache.
- **Opteron**: 4 AMD Opteron 6276 processors and 128 GiB of RAM running Linux kernel 4.15.0-96. Each socket has 8 2-SMT cores, totaling 32 cores. 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.

On the Xeon machine, we run the applications using 32, 64, and 96 threads, and on Opteron, 16, 32 and 64. We chose different MI for the machines. For the Opteron machine, a MI of 50,000 was enough to get necessary information for the heuristic. On Xeon, since we used higher thread numbers, we used a MI of 100,000.

To evaluate our proposal, each application was executed 10 times using different mapping strategies:
Linux is the default Linux CFS scheduler used as our baseline.
Compact places threads on sibling cores that share all cache levels, thus potentially reducing the data access latency if neighboring threads communicate [15].
Scatter distributes threads across different processors, avoiding cache sharing, thus, reducing memory contention [15].
Round-Robin is a mix between compact and scatter, where only the last level of cache is shared [15].
Static-SharingAware (SSA) is based on our previous work [19]. We first trace the behavior of each application in order to determine the communication pattern. Then, we calculate the new thread mapping offline using `TopoMatch` and re-execute the application with this static mapping. This approach has no runtime overhead, but is not able to handle changes in the application behavior (during execution or if

the behavior is different between executions).
STMap is the mechanism proposed in this paper (Sect. III).

The Compact, Scatter, and Round-Robin mappings were calculated using the `hwloc` library. The overhead added to calculated the mapping is approximately 70 ms. With exception of *STMap* and *Linux*, the mappings are applied when a thread calls the `stm_init_thread` function of the `TinySTM` library, which informs the runtime that the thread will perform transactional operations.

B. Results on the Xeon Machine

Fig. 2 shows the execution time (in seconds) on the Xeon machine. Also, each bar shows the average and standard deviation. The Static-SharingAware approach will be abbreviated as SSA in the discussion. When discussing the distinct address accessed and commit and abort ratios, these metrics are based on when the MI was triggered. Percentages of improvement are always compared to Linux, if not specified.

Bayes is an application with high contention, spending lots of time inside transactions [9]. It accesses a low number of distinct addresses. This application had a performance increase with STMap in 32 (17.6%) and 96 threads (8.9%) compared to Linux. This means that this application has a communication pattern that is suitable for the sharing-aware approach.

Genome has little contention and spends lots of time inside transactions [9]. It access a large number of distinct addresses, roughly twice the $da\_threshold$ metric (Sect. III-C). Since it has little contention, i.e, the abort ratio is low, the online mechanism was disabled correctly. Although there is a large difference between SSA and STMap, scatter performed better. Looking at 96 threads SSA had a performance loss of 26% whereas in STMap we could reduce this loss to 7.1%, by disabling the mechanism.

Intruder has high contention and spends medium time inside transactions [9]. Results with 32 threads were similar for Compact and SSA. The speedup achieved with STMap was 25% in this configuration. However, as the number of threads grows, the advantages of a sharing-aware mechanism diminish. With 64 threads the speedup was still the best, followed by Linux, but with 96 threads we had a performance loss of 16%. This indicates that *Intruder* has a sharing pattern that is strongly related to the number of threads.

Kmeans has little contention and spends little time inside transactions [9]. Due to the low contention, we can expect similar results as in *Genome*. However, this application accesses a very small amount of distinct addresses (roughly 20), and all accesses are made to these addresses. This shows that we cannot rely only on commit and abort ratios. Using 64 and 96 threads we achieved good results with STMap, with a speedup of 30.3% and 21.5%, respectively. Also, it is worth noting that with 96 threads our results were even better than the SSA mechanism. This can be explained because the application has a different sharing pattern on each execution. Thus, only an online mechanism can adapt to this behavior. In 32 threads SSA achieved a speedup of 58.3%. Hence, we excepted similar results with STMap. However, we noticed an issue with the MI of 100,000 used on this machine. Only in the case of 32 threads, this MI was too high, and the application never triggered the mechanism. This explains the low gains of STMap with 32 threads.

Labyrinth has high contention and spends lots of time inside transactions [9]. However, it accesses a large number of distinct addresses and the mechanism was correctly disabled here. Using the SSA approach we had a performance loss of 53.5% with 32 threads. However, the STMap mechanism performed similar to Linux, in all configurations. Also, it is worth noting that all mappings performed similarly, except Compact and SSA. This application shows the importance of having a heuristic to disable the mechanism, if the mechanism can predict that the final performance would be worse.

Ssca2 has little contention, spending little time inside transactions [9]. The best mapping was SSA in all configurations. This application has similar characteristics as *Genome*, hence, the mechanism was disabled and the results achieved with STMap were similar to Linux.

Vacation has medium contention and spends lots of time inside transactions [9]. This application has complex characteristics that are harder to predict in all thread configurations. We have three distinct cases. Using 32 threads, SSA had a speedup of 5.2% over Linux. However, due to the overhead of the mechanism, STMap was similar to Linux. Using 64 threads, the abort and commit ratio was not deterministic. We had roughly half of the times the mechanism disabled, due to a higher commit ratio. However, it is possible to see in the error bar, that sometimes the mechanism was enabled, and performed better than SSA. In 96 threads, the mechanism was incorrectly disabled all the time.

Yada has medium contention and spends lots of time inside transactions [9]. It accesses a medium amount of distinct addresses. The results in all configurations of threads were similar, with Compact and SSA performing better than other mappings. Nevertheless, STMap performed better than Linux in all thread configurations, with speedups of 47.1%, 12.7%, and 48.2%, respectively.

Red-black tree has a communication pattern where threads communicate often with their neighbors. Thus, Compact has a good performance. Nevertheless, using 96 threads, STMap had the best speedup of 58% over Linux.

Hashmap has a communication pattern similar to *Red-black tree*, but STMap did not result in the highest gains using 32 and 64 threads. Specifically with 64 threads, in some runs the $da\_threshold$ was higher, and since the commit ratio of this application is high, the mechanism was disabled (Equation 1). The execution time varies between 19 (mechanism enabled) to 71 seconds. This explains the large error bar. However, using 96 threads the mechanism was enabled in all executions, achieving a speedup of 77.7% over Linux.

C. Results on the Opteron machine

Fig. 3 shows the performance results on the Opteron machine. Since we already discussed the characteristics of the

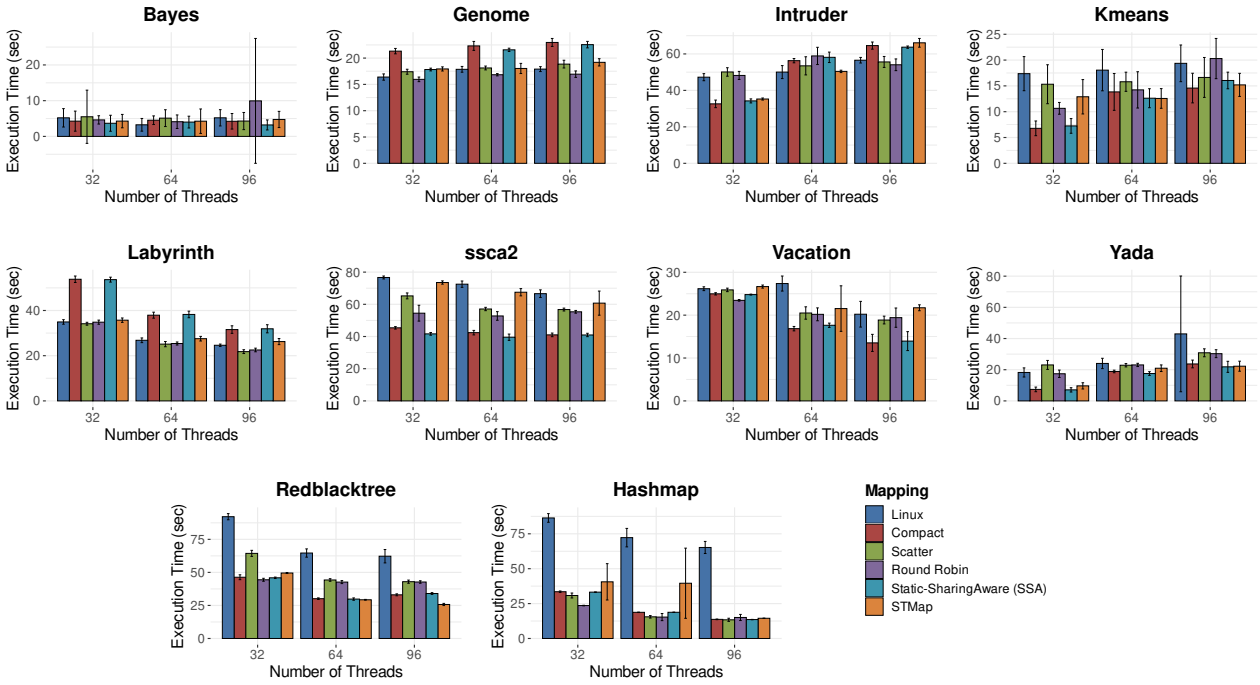


Fig. 2: Execution time on the Xeon Machine.

benchmarks in the previous Section, we will only discuss the performance results.

Bayes had a performance increase in 64 threads with STMap (18.7%) compared to Linux. For 16 and 32 threads, the results were similar to Linux.

Genome had a similar behavior as on the Xeon machine, and STMap was disabled correctly. Using SSA, we had a performance loss with 32 and 64 threads. With 64 threads, STMap achieved the best results, roughly equal to Linux.

In *Intruder* the results were similar for Compact and SSA, both for 32 and 64 threads. Nevertheless, the speedup achieved with STMap with 16 threads was the best for the Opteron machine (40.6%). Using 32 threads, the speedup was 36.9% compared to Linux.

In *Kmeans*, contrary to the Xeon machine, the STMap mechanism was not disabled and high gains were achieved, mainly with 32 threads (38.7%).

In *Labyrinth*, the behavior was similar to Xeon. Since it accesses a large number of distinct addresses (approx. 14,000) and the commit ratio is higher, the mechanism was correctly disabled. SSA resulted in performance losses, mainly with 16 and 32 threads. On the other hand, the STMap mechanism performed similarly to Linux.

Ssca2 is another application with similar behavior on both machines. Unfortunately, since the best mapping was SSA we expect good results with STMap as well. However, as on the Xeon machine, STMap was incorrectly disabled for this application.

Vacation accesses a large number of distinct addresses and the abort ratio was slightly greater than the commit ratio in this machine. Thus, the mechanism was correctly disabled for 32 threads. However, no mapping had a big impact on the performance.

Yada accesses a medium number of distinct addresses (roughly 2,000). However, contrary to the Xeon machine, we did not observe big differences between the mappings, with exception of Linux with 64 threads. Nevertheless, SSA and STMap had similar results as Linux.

In the last two benchmarks, *Hashmap* and *Red-black tree*, SSA was the best mapping. STMap had slightly smaller gains compared to SSA that can be explained by the overhead in performing the mapping online.

D. Discussion

As expected, creating a unique heuristic that fits in all cases is a challenge. The main drawback of the proposed STMap mechanism appeared with *Ssca2*. On both machines, STMap was disabled by the heuristic, but we had good results using the SSA approach. On the other hand, in *Genome* and *Labyrinth* the mechanism was correctly disabled, avoiding a greater performance loss if the mechanism was enabled. Analyzing the results, sharing-aware mapping in STM depends on a low number of distinct addresses with a lot of accesses in these addresses. Also, applications with low and medium contention had higher gains.

It is worth noting that only transactional operations were tracked to determine the sharing behavior. The intuition is that if a memory location is shared by more than one thread, it will be protected by transactional operations.

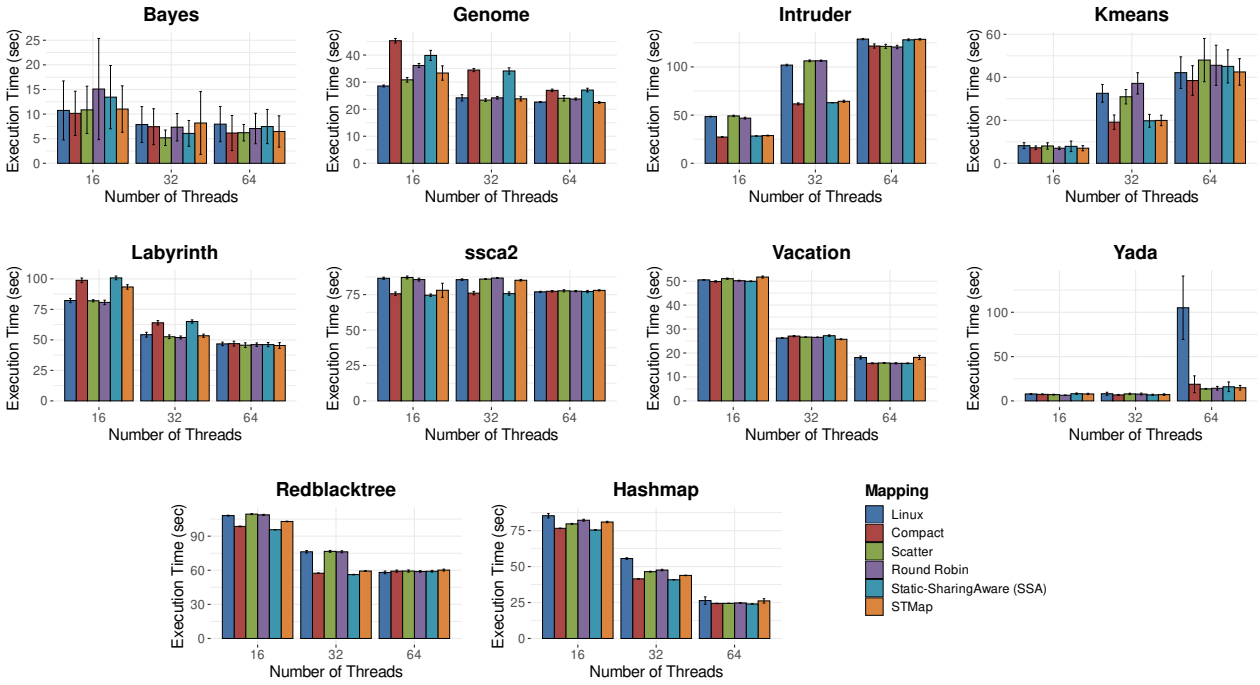


Fig. 3: Execution time on the Opteron Machine.

On the Opteron machine, the best speedup over Linux was achieved by *Yada* using 64 threads (85.8%) and by Intruder using 16 threads (40.6%). On the Xeon machine, the highest speedup was achieved by *Hashmap*, achieving gains of 77.7% over Linux. TABLE I shows the average speedup of each mechanism over Linux, taking into consideration all applications and thread configurations.

TABLE I: Average speedup of each mechanism over Linux.

Machine	Compact	Scatter	RoundRobin	SSA	STMap
Xeon	15.88%	13.14%	14.50%	22.32%	17.57%
Opteron	7.96%	5.86%	3.15%	6.50%	9.14%

V. Related work

Many proposals to improve STM performance rely on the use of schedulers. Their main objective is serializing transactions when conflicts are predicted or detected. One of the first papers to propose a transaction scheduler was *Adaptive transaction scheduling* (ATS) [20]. To predict future conflicts between transactions, it uses a metric calculated for each thread called *contention intensity*. Then, many scheduling techniques were proposed, for instance, [21] and [22]. In [3], a survey of scheduling techniques for STM is presented.

Beyond schedulers, many proposals focus on the use of *concurrency control*. The main idea is to limit the total number of threads running concurrently. They argue that an excessive number of threads can hurt the performance in high contention workloads, mainly because of the high number of aborts. One of the first papers using this idea was *Adaptive Concurrency Control* (ACC) [23]. Then, many concurrency controls were proposed, for instance, [24] and [25].

Newer proposals focus on the use of *thread mapping*, to better exploit the machine resources, such as caches. In [16], an affinity-aware thread migration was proposed. The main objective is reducing cross-die cache invalidation, by keeping transactions that conflict often, because they are accessing the same shared data, in sibling cores with shared caches. Each time that a transaction aborts, the conflicting transaction is detected and the probability of conflict between them is updated in a matrix. When the mechanism is triggered, a pair of threads is chosen to migrate between sockets. One drawback of this approach is that it is focused on machines with two sockets. The sharing-aware approach presented in this paper can be more accurate since it is based on each transactional read and write. Furthermore, the thread mapping is global, i.e., taking into consideration all threads.

Castro et al. [15] use a different thread mapping strategy, based on the abort ratio of transactional applications. Using the abort ratio and a threshold, in a predetermined amount of time, the thread mapping is dynamically changed between scatter, compact, or round-robin. In Zhou et al. [26], a concurrency control mechanism to define dynamically the best number of threads allowed to run concurrently is proposed. The main objective is to maximize the throughput. When the number of threads running is less than half of the total number of cores, another mechanism is triggered, and the thread mapping is changed to round-robin. Monitoring the new throughput, the mapping could be changed to Linux default, compact, or scatter. In both proposals, the actual memory access behavior

of the applications is not taken into consideration.

Góes et al. [27] focused on STM applications that have a worklist pattern and proposed a mechanism for improving memory affinity that relies on the use of static page allocation (bind or cyclic) and data prefetching by using helper threads. The mechanism was implemented in the `OpenSkel` framework. STM applications need to be rewritten with this framework to be able use the memory affinity improvements.

Our previous work [19] discussed extracting the memory access behavior of STM applications using a static approach. The mechanism proposed in this paper uses a dynamic approach, detecting the memory access behavior and calculating the improved mapping while the application is running.

CDSM [8] is a kernel module to detected communication patterns of parallel applications and migrate threads according to their memory affinity. Differently from our approach, CDSM tracks the memory of all running applications instead of the only STM ones.

VI. Conclusion

Mapping threads and data according to their memory access behavior is a well-known technique for improving the performance of applications. This technique is called sharing-aware mapping, and the objective is to make communication and cache usage more efficient. In this paper, we presented *STMap*, an online mechanism to extract the memory access behavior of STM applications and use this information to calculate a sharing-aware mapping of threads to cores. The main advantage of our mechanism is that no prior knowledge is necessary, all phases are executed while the application is running. Also, applications are not modified, only the STM system. Using `STAMP` applications and synthetic benchmarks, STMap achieved performance gains of up to 77% on a Xeon machine (17.5% on average) and 85.8% on an Opteron machine (9.1% on average) compared to the Linux scheduler. For the future, we intend to improve our heuristic to disable the mechanism, using other metrics, such as caches misses or deeper analysis of the communication matrix. Also, we intend to add data to our mapping mechanism.

References

[1] H. Grahn, "Transactional memory," *J. Parallel Distrib. Comput.*, vol. 70, no. 10, pp. 993–1008, Oct. 2010.

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

[3] 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.

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

[5] 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.

[6] D. Beniamine, M. Diener, G. Huard, and P. O. A. Navaux, "TABARNAC: Visualizing and resolving memory access issues on NUMA architectures," in *Proceedings of the 2nd Workshop on Visual Performance Analysis*, ser. VPA '15. ACM, 2015, pp. 1:1–1:9.

[7] F. Trahay, M. Selva, L. Morel, and K. Marquet, "NumaMMA - NUMA MeMory Analyzer," in *Proceedings of the 47th International Conference on Parallel Processing - ICPP 2018*. ACM Press, 2018.

[8] M. Diener, E. H. Cruz, P. O. Navaux, A. Busse, and H.-U. Heiß, "Communication-aware process and thread mapping using online communication detection," *Parallel Computing*, vol. 43, pp. 43–63, 2015.

[9] 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.

[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] C. Bordage and E. 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*, ser. CCGrid '18. IEEE Press, 2018, p. 523–532.

[12] Z. Wang and A. C. Bovik, "Mean squared error: Love it or leave it? a new look at signal fidelity measures," *IEEE Signal Processing Magazine*, vol. 26, no. 1, pp. 98–117, 2009.

[13] E. Jeannot, "TopoMatch: Process mapping algorithms and tools for general topologies," https://gitlab.inria.fr/ejeannot/topomatch, 2020, last accessed 15th, April 2020.

[14] E. Jeannot, G. Mercier, and F. Tessier, "Process placement in multicore clusters: Algorithmic issues and practical techniques," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 4, p. 993–1002, 2014.

[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] 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.

[17] 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.

[18] 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.

[19] D. P. Pasqualin, M. Diener, A. R. Du Bois, and M. L. Pilla, "Thread affinity in software transactional memory," in *19th Int. Symposium on Parallel and Distrib. Comput.* IEEE CS, July 2020.

[20] 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.

[21] 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.

[22] H. Rito and J. a. Cachopo, "Adaptive transaction scheduling for mixed transactional workloads," *Parallel Comput.*, vol. 41, p. 31–49, 2015.

[23] 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.

[24] 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.

[25] K. Ravichandran and S. Pande, "F2C2-STM: Flux-based feedback-driven concurrency control for STMs," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14. USA: IEEE Computer Society, 2014, p. 927–938.

[26] 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.

[27] 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.