Dynamic Thread Mapping Based on Machine Learning for Transactional Memory Applications

Márcio Castro[1], Luís Fabrício Wanderley Góes[2],
Luiz Gustavo Fernandes[3], and Jean-François Méhaut[1]

[1] INRIA - CEA - LIG Laboratory - Grenoble University,
ZIRST 51, avenue Jean Kuntzmann, 38330 Montbonnot Saint Martin, France
{marcio.castro,jean-francois.mehaut}@imag.fr
[2] Department of Computer Science - Pontifical Catholic University of Minas Gerais,
Av. Dom José Gaspar, 500, Belo Horizonte, MG, Brazil
lfwgoes@pucminas.br
[3] PPGCC - Pontifical Catholic University of Rio Grande do Sul,
Av. Ipiranga, 6681 - Prédio 32, Porto Alegre, RS, Brazil
luiz.fernandes@pucrs.br

Abstract. Thread mapping is an appealing approach to efficiently exploit the potential of modern chip-multiprocessors. However, efficient thread mapping relies upon matching the behavior of an application with system characteristics. In particular, Software Transactional Memory (STM) introduces another dimension due to its runtime system support. In this work, we propose a dynamic thread mapping approach to automatically infer a suitable thread mapping strategy for transactional memory applications composed of multiple execution phases with potentially different transactional behavior in each phase. At runtime, it profiles the application at specific periods and consults a decision tree generated by a Machine Learning algorithm to decide if the current thread mapping strategy should be switched to a more adequate one. We implemented this approach in a state-of-the-art STM system, making it transparent to the user. Our results show that the proposed dynamic approach presents performance improvements up to 31% compared to the best static solution.

Keywords: transactional memory, dynamic thread mapping, machine learning.

1 Introduction

Thread mapping is an appealing approach to efficiently exploit the potential of modern chip-multiprocessors by making better use of cores and memory hierarchy. It allows multithreaded applications to amortize memory latency and/or reduce memory contention. However, efficient thread mapping relies upon matching the behavior of an application with system characteristics.

Software Transactional Memory (STM) appears as a promising concurrency control mechanism for those modern chip-multiprocessors. It allows programmers to write parallel code as transactions, which are guaranteed to execute atomically and in isolation regardless of eventual data races [3,9]. At runtime, transactions are executed speculatively and the STM runtime system continuously keeps track of concurrent accesses and detects conflicts. Conflicts are then solved by re-executing conflicting transactions.

C. Kaklamanis et al. (Eds.): Euro-Par 2012, LNCS 7484, pp. 465–476, 2012.

However, due to its runtime support, applications can behave differently depending on the characteristics of the underlying STM system. Thus, the prediction of a suitable thread mapping strategy for a specific application/STM system becomes a daunting task.

Our previous work focused on a machine learning-based approach to statically infer a suitable thread mapping strategy for transactional memory applications [2]. This means that the predicted thread mapping strategy is applied once at the beginning and does not change during the execution of the application. We demonstrated that this approach improved the performance of all STAMP applications [10], since most of the transactions within each application usually have very similar behavior.

We have constantly seen efforts for a wider adoption of Transactional Memory (TM). For instance, the latest version of the GNU Compiler Collection (GCC 4.7) now supports TM primitives and new BlueGene/Q processors have hardware support for TM. Moreover, Intel recently released details of the Transactional Synchronization Extensions (TSX) for the future multicore processor code-named "Haswell". Thus, it is expected that more complex applications will make use of TM in a near future. In those cases, static thread mapping will no longer improve the performance of those applications, emerging the necessity of dynamic or adaptive approaches.

In this paper, we propose a dynamic approach to do efficient thread mapping on STM applications composed of more diverse workloads. These workloads may go through different execution phases, each phase with potentially different transactional characteristics. At runtime, we gather useful information from the application, STM system and platform at specific periods. At the end of each profiling period, we rely on a decision tree previously generated by a Machine Learning (ML) algorithm to decide if the current thread mapping strategy should be switched to a more adequate one. This dynamic approach was implemented within TinySTM [5] as a module, so the core of TinySTM remains unchanged and it is transparent to the user. Our results show that the dynamic approach is up to 31% better than the best static thread mapping for those applications.

The rest of this paper is organized as follows. Section 2 further describes STM and our previous work. In Section 3, we propose our dynamic thread mapping mechanism. Section 4 evaluates our dynamic thread mapping on several applications. Finally, Section 5 presents related work and Section 6 concludes.

2 Background

2.1 Software Transactional Memory

Transactional Memory is an alternative synchronization solution to the classic mechanisms such as locks and mutexes [9]. It removes from the programmer the burden of correct synchronization of threads on data races and provides an efficient model for extracting parallelism from the applications.

Transactions are portions of code that are executed atomically and with isolation. Concurrent transactions *commit* successfully if their accesses to shared data did not conflict with each other; otherwise some of the conflicting transactions will *abort* and none of their actions will become visible to other threads. Conflicts can be detected

during the execution of transactions when the TM system uses an *eager conflict detection policy* whereas they are only detected at commit-time when the system uses a *lazy conflict detection policy*.

When a transaction aborts, the runtime system *rollbacks* some of the conflicting transactions. The choice among the conflicting transactions is done according to the *conflict resolution policies* implemented in the runtime system. Two common alternatives are to squash one of the conflicting transactions immediately (*suicide* strategy) or to wait for a time interval before restarting the conflicting transaction (*backoff* strategy).

Transactional Memory can be software-only, hardware-only or hybrid. In this work we are interested in STM since hardware and hybrid solutions are not yet available in commercial processors. This allows us to carry out experiments in current platforms without relying on simulations.

2.2 Static Thread Mapping Based on Machine Learning

Our previous work proposed a machine learning-based approach to predict a suitable thread mapping for TM applications [2]. It was composed of the following steps (Figure 1). Firstly, we profiled several TM applications from the STAMP benchmark suite [10] considering characteristics from the application, STM system and platform to build a set of input instances. Then, a Decision Tree Learning method (ID3) [11] was fed with these input instances and *trained*. The ID3 algorithm outputted a decision tree (predictor) capable of infering a thread mapping strategy for new unobserved instances.

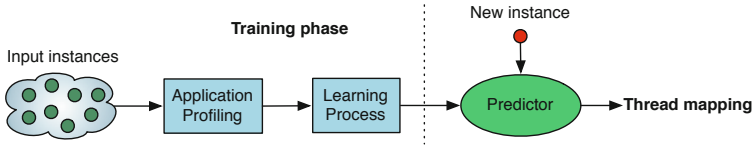

Fig. 1. Overview of our machine learning-based approach

We evaluated the performance of all TM applications from STAMP when applying the predicted thread mapping strategies *statically*. This means that the strategy is applied at the beginning and remains unchanged during the whole execution of the application. Our results showed that our approach usually makes correct predictions [2]. However, a deeper analysis of STAMP applications revealed that most of them do not have multiple phases with different transactional characteristics. For the upcoming complex workloads, there may not exist a single best thread mapping strategy that delivers the best performance for all phases. In the next section we present our solution to tackle this problem, which employs a dynamic approach adapted for more complex applications.

3 Dynamic Thread Mapping for Transactional Memory

As we previously stated, we will naturally face more complex TM applications due to the wider adoption of TM. These applications will probably have multiple execution

phases with a different transactional behavior in each phase. Thus, we need a more dynamic thread mapping approach able to identify these different phases and switch to a more adequate thread mapping strategy during the execution. In the following sections, we explain the basic concepts of our dynamic approach as well as its implementation within a state-of-the-art STM system.

3.1 Proposed Approach

Our dynamic thread mapping approach is based on the fact that the performance of a TM application is not only governed by its characteristics but also by the characteristics of the TM system and platform. Those characteristics must be taken into account to choose a thread mapping strategy adapted to behavior of the workload. Thus, we consider the following criteria that have an important impact on the performance of TM applications:

- **Transactional time ratio:** fraction of the time spent inside transactions to the total execution time;
- **Abort ratio:** fraction of the number of aborts to the number of transactions issued (aborted + committed);
- **Conflict detection policy:** eager or lazy;
- **Conflict resolution policy:** suicide or backoff;
- **Last-level cache miss ratio:** fraction of the number of cache misses to the number of accesses on the last-level cache.

We considered these criteria while profiling the STAMP applications to build a *thread mapping predictor* as briefly described in Section 2.2. We trained the ID3 learning algorithm with two sets of input instances. The difference between them comes from the complexity of the memory hierarchy of the underlying platform. The predictor is represented in Figure 2.

The subtree on the left considers a single level of shared L2 caches whereas the subtree on the right considers a more complex memory hierarchy with two levels of shared caches (L2 and L3). Internal nodes represent our criteria (rectangles). Leaves represent the thread mapping strategy to be applied (rounded rectangles).

The predictor chooses a thread mapping strategy among four possible configurations: *scatter, compact, round-robin* and *linux*. *Scatter* distributes threads across different processors avoiding cache sharing between cores in order to reduce memory contention. In contrast, *compact* places threads on sibling cores that share all levels of the cache hierarchy. The *round-robin* strategy is an intermediate solution in which threads share higher levels of cache (*i.e.*, L3) but not the lower ones (*i.e.*, L2). Finally, *linux* is the default scheduling strategy implemented by the operating system.

Since most of the considered characteristics can vary during the execution of applications composed of several phases, they need to be profiled at runtime. We thus use profiling to gather the information needed by the predictor at specific periods. We specify two periods: the profiling period and the interval between profilings. These values are specified by the number of committed transactions instead of time. This guarantees that our measures occur when transactions are being executed. We use a hill-climbing

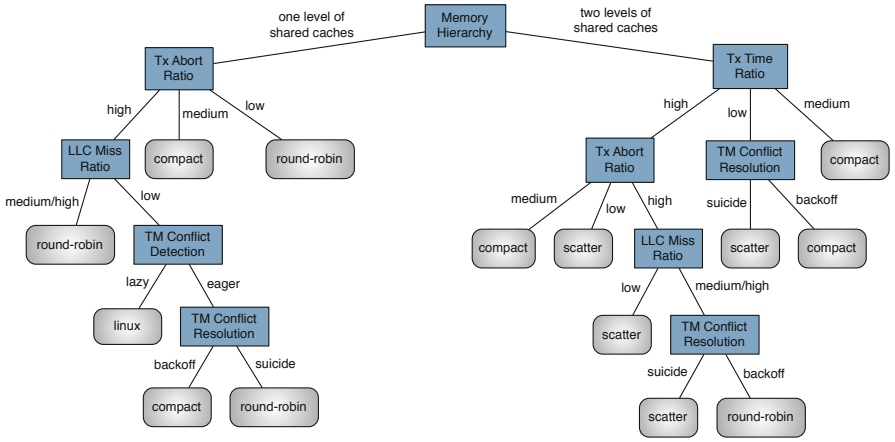

Fig. 2. Thread mapping predictor based on machine learning

strategy to adapt those values during the execution. We start with short periods and we double them each time the predicted thread mapping strategy was not changed. This is done until a maximum interval size is reached. When the thread mapping strategy is changed due to a phase transition, we reset them to their initial values and the hill-climbing strategy is restarted.

3.2 Implementation

For our solution to be transparent to users, we decided to implement it within a STM system. We chose TinySTM [5] among other STM systems because it is lightweight, efficient and its implementation has a modular structure that can be easily extended with new features. Figure 3 shows the organization schema of TinySTM and as well as our dynamic thread mapping module and its main components.

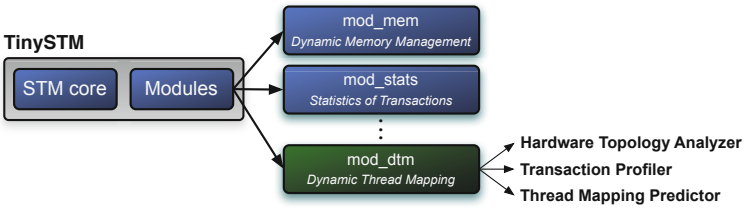

Fig. 3. Implementation of our dynamic thread mapping in TinySTM

Basically, TinySTM is composed of a STM core in which most of the STM code is implemented, and some additional modules. These modules implement basic features such as the dynamic memory management (`mod_mem`) and transaction statistics

(`mod_stats`). We added a new module called `mod_dtm` that extends TinySTM to perform dynamic thread mapping transparently. Our module combines the following three main components:

Hardware topology analyzer uses the Hardware Locality (hwloc) library [1] to gather useful information from the underlying platform topology (*i.e.*, the hierarchy of caches and how they are shared among the cores). Such information is used to correctly apply the thread mapping strategies.

Thread mapping predictor relies on the decision tree shown in Figure 2 to predict the thread mapping strategy. At the end of each profiling period, the tree is traversed using the profiled information from the *transaction profiler* and the resulting thread mapping strategy is then applied.

Transaction profiler performs runtime profiling during specific periods to gather information from hardware counters and transactional basic statistics. Its pseudo-code is depicted in Figure 4. The *cache miss ratio* is obtained through the Performance Application Programming Interface (PAPI) [12] to access hardware counters. We maintain two counters to calculate the *abort ratio* (named Aborts and Commits). The *transactional time ratio* is an approximation obtained by measuring the time spent inside and outside transactions.

```
// on transaction start
if is profiling period then
    if first tx in this period then
        StartPapi(LLCAccess, LLCMiss);
        ProfileTime ← GetClock();
    end
    TxTime ← GetClock();
end

// on transaction abort
if is profiling period then
    Aborts ← Aborts + 1;
end
```

```
// on transaction commit
if is profiling period then
    TxTime ← GetClock() − TxTime;
    TotalTxTime ← TotalTxTime + TxTime;
    Commits ← Commits + 1;
    if last tx in this period then
        StopPapi(LLCAccess, LLCMiss);
        ProfileTime ← GetClock() − ProfileTime;
        TotalNonTxTime ← ProfileTime − TotalTxTime;
        ThreadMapping ← TMPredictor();
        ResetAllCounters();
    end
end
```

Fig. 4. Transaction profiler pseudo-codes

TinySTM allows the inclusion of user-defined extensions. In our case, we instrumented three basic TM operations that are called when transactions start (`start`), when they are rollbacked in case of conflicts (`abort`) and when they finish successfully (`commit`). Thus, every call to these operations is intercepted by our module, which executes the *transaction profiler* during the profilling periods and calls the *thread mapping predictor* to switch the thread mapping strategy when necessary.

When a TM application is executed, only one thread among all concurrent running threads is chosen to be the *transaction profiler*. The reason for that is threefold: (i) it considerably reduces the intrusiveness on the overall system, so the behavior of the application is not changed; (ii) we do not need to use extra synchronization mechanisms to guarantee reliable measures among concurrent threads; and (iii) most workloads of current TM applications are uniformly distributed among the threads. However, our

implementation can be adapted to gather information from all threads. This may be necessary for non-SPMD applications, where different threads execute different flows of control.

4 Experimental Evaluation

In this section, we demonstrate that our dynamic thread mapping can benefit from applications composed of multiple execution phases with potentially different transactional behavior on each one. First, we describe our experimental setup as well as the set of characteristics we considered to create TM applications composed of multiple phases. Afterwards, we compare our performance gains with static solutions. Finally, we present a deeper analysis of our mechanism.

4.1 Experimental Setup

Since most of the transactions within each STAMP application usually have very similar behavior, they are not suitable for the evaluation of our dynamic thread mapping approach. For this reason, we used EigenBench [8] to create new TM applications with different phases. This micro-benchmark allows a thorough exploitation of the orthogonal space of TM applications characteristics.

Varying all possible orthogonal TM characteristics involves a high-dimensional search space [8]. Thus, we decided to vary 4 out of 8 orthogonal characteristics that govern the behavior of TM applications. We used the first three (transaction length, contention and density) to create a set of workloads (Table 1). Since we assume two possible discrete values for each one, we can create a total of 2^3 distinct workloads (named $W_1, W_2, \ldots, W_8$) by combining those values. It is important to mention that these values were obtained after an empirical study based on several experiments with different configurations of TinySTM (conflict detection and resolution policies) and EigenBench parameters. The fourth orthogonal characteristic is *concurrency* and it is further discussed in Section 4.3.

Table 1. TM orthogonal characteristics used to compose our set of workloads

Characteristic	Definition	Values
Tx Length	number of shared accesses per transaction	short (≤ 64) long (≥ 128)
Contention	probability of conflict	low-conflicting ($< 30\%$) contentious ($\geq 30\%$)
Density	fraction of the time spent inside transactions to the total execution time	sparse ($< 80\%$) dense ($\geq 80\%$)
Concurrency	number of concurrent threads/cores	2 – 16

We conducted our experiments on a multi-core platform based on four six-core 2.66GHz Intel Xeon X7460 processors and 64 GB of RAM running Linux 2.6.32. Each processor has 16MB of shared L3 cache and each group of two cores shares a L2 cache (3MB). TinySTM and all applications were compiled with GCC 4.4.5 using `-O3`. All results in the following sections are based on arithmetic means of 30 runs.

4.2 Dynamic Thread Mapping vs. Static Thread Mapping

Our first set of experiments explores the effectiveness of our dynamic thread mapping in comparison to the thread mapping strategies individually. We derived a set of applications from the 8 distinct workloads discussed in Section 4.1. We fixed the number of phases to 3, thus each application will be composed of three workloads. Therefore, all possible applications composed of three distinct workloads is determined by the number of k-combinations from a given set of n elements, *i.e.*, $C_k^n = C_3^8$, which results in 56 applications (named $A_1, A_2, \ldots, A_{56}$). Thus, the set of applications can be represented as follows: $A_1 = \{W_1, W_2, W_3\}$, $A_2 = \{W_1, W_2, W_4\}$, ..., $A_{56} = \{W_5, W_6, W_7\}$. Phases (workloads) are parallelized using Pthreads and there is no synchronization barrier between phases, *i.e.*, threads may not be computing the same workload at the same time.

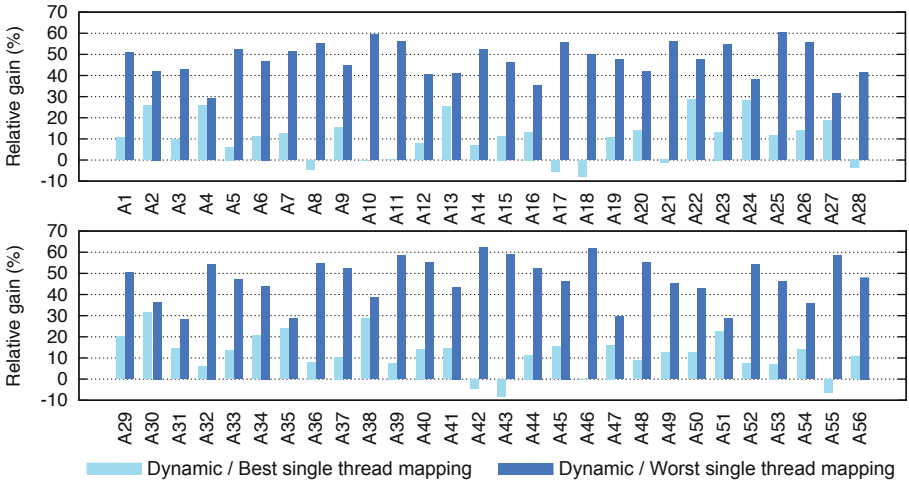

Fig. 5. Relative gains of our dynamic thread mapping compared to the best and worst single thread mappings. We considered applications composed of 3 phases (A_1 to A_{56}).

We ran all the applications with each one of the static thread mappings (*compact, round-robin* and *scatter*), the Linux default scheduling strategy and our dynamic approach. Figure 5 presents the relative gains of our dynamic thread mapping when compared to the **best** and **worst** single thread mappings. The relative gain is given by $1 - \overline{x}_d \div \overline{x}_s$, where $\overline{x}_d$ and $\overline{x}_s$ are mean execution times of 30 executions using the dynamic and the best/worst single thread mapping, respectively. Thus, positive values mean performance gains whereas negative values mean performance losses. All applications were executed with 4 threads and TinySTM was configured with lazy conflict detection and backoff conflict resolution.

We can draw at least two important conclusions from these results. Firstly, the thread mapping strategy had an important impact on the performance. This can be easily observed when comparing the relative gains between the best and worst single thread mappings. Secondly, our dynamic thread mapping usually improved the performance

of the applications by switching to an adequate thread mapping strategy in each phase. We achieved performance gains up to 31% and 62%, when comparing to the best and worst single thread mappings respectively. However, our dynamic thread mapping did not deliver performance improvements on 3 applications and presented some performance losses in 8 applications when comparing with the best single thread mapping strategy. In the case of A_{10}, A_{11} and A_{46}, a single thread mapping strategy (*compact*) was best for all phases, thus we cannot expect performance improvements by using our dynamic approach. The performance losses were due to wrong decisions of the predictor, which did not select the best thread mapping strategy on all phases. The maximum performance loss was about 8% (A_{43}). One reason for that may come from the characteristics that we take into account in training phase and profiling. We leave the discussion of other possible characteristics to enrich the predictions to future work.

4.3 Varying Concurrency

Our second set of experiments focuses on the performance impacts of the thread mapping strategies when varying the number of threads. We selected 4 interesting cases. Cases 1 and 2 are applications that presented a single best thread mapping strategy for all thread counts. Cases 3 and 4 are applications whose the best single thread mapping varied according to the number of threads.

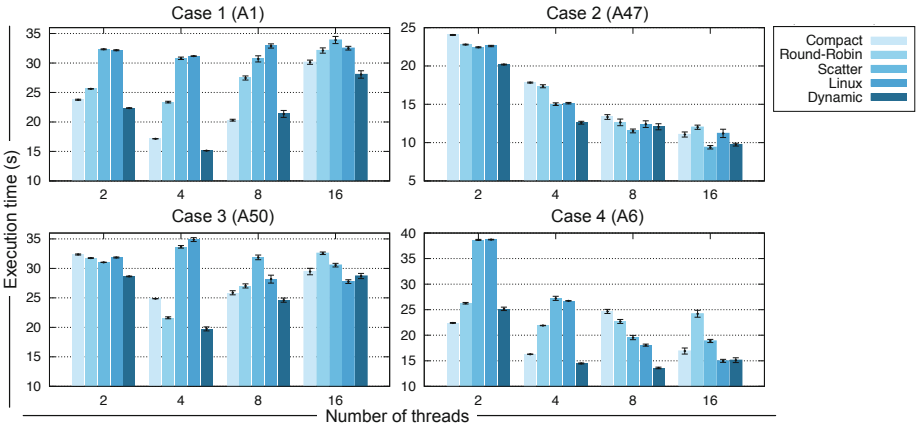

Fig. 6. Execution times when varying the number of threads

Figure 6 compares the execution times of the four single thread mapping strategies with our dynamic thread mapping mechanism. Results represent mean execution times of 30 executions with 95% confidence intervals. We do not consider more than 16 threads for two reasons: (i) placing threads on different cores when all available cores are used does not impact the overall performance because the applications tend to communicate uniformly, and (ii) most of our workloads did not scale beyond 16 threads.

In Case 1, the best single thread mapping for all thread counts was *compact* whereas in Case 2 it was *scatter*. In both cases our dynamic thread mapping presented lower execution times for most of the thread counts. Case 3 represents a scenario in which the best

single thread mapping strategy relied on the number of threads (*scatter*, *round-robin*, *compact* and *linux* with 2, 4, 8 and 16 threads respectively). In case 4, we observed that *compact* was best for low thread counts whereas *linux* was best for high thread counts. In both cases 3 and 4, our dynamic thread mapping usually resulted in better results than single thread mappings.

4.4 Dynamic Thread Mapping in Action

In order to observe how our dynamic thread mapping reacts when it encounters several different phases, we created a single application composed of all the 8 distinct workloads. We then executed this application with our dynamic thread mapping while tracing the information obtained by the *transaction profiler* at the end of each profiling period. Figure 7 shows the variance of the profiled metrics during the execution with 4 threads. Vertical bars represent the intervals in which each thread mapping strategy was applied.

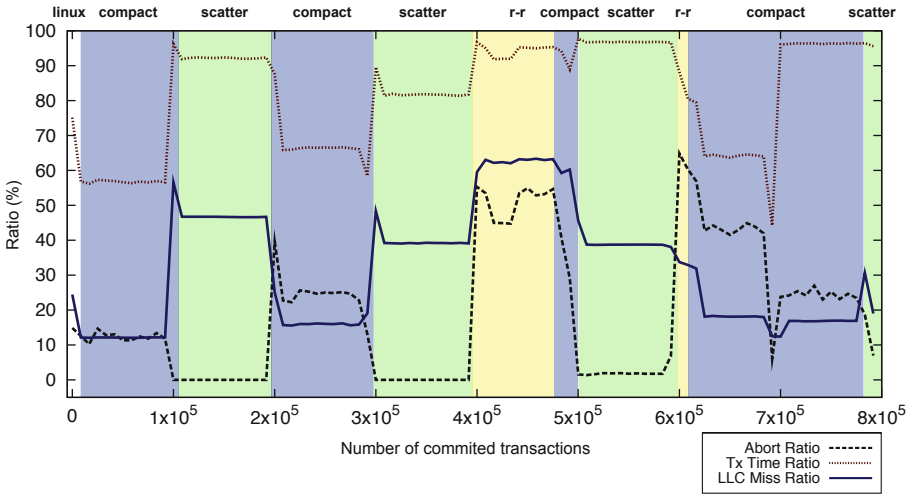

Fig. 7. Profiled metrics during the execution of an application with 8 phases

At the beginning, our dynamic thread mapping mechanism applies *linux* as its default strategy and profiles some transactions. After the first profiling period, the predictor decided to apply *compact* and did not switch to another strategy until it reached a different phase near 1×10^5. At this point, the predictor switched to *scatter*. Overall, the predictor detected more than 8 phases due to the variance of some profiled metrics but it still detected correctly the 8 main phase changes, reacting by applying a suitable thread mapping strategy for each phase. We can also observe that the variance of the profiled metrics confirms the fact that the 8 workloads have distinct characteristics.

5 Related Work

Thread Mapping. In [15], the authors presented a process mapping strategy for MPI applications. The strategy used a graph partitioning algorithm to generate an appro-

priate process mapping for an application. The proposed strategy was then compared with *compact* and *scatter*. In [4], two thread mapping algorithms were proposed. These algorithms relied on memory traces extracted from benchmarks to find data sharing patterns between threads. These patterns were extracted by running the workloads on a simulator. The proposed approach was compared to *compact*, *scatter* and other strategies. In [7], the authors proposed a dynamic thread mapping strategy for regular data parallel applications implemented with OpenMP. The strategy considered the machine description and the application characteristics to map threads to processors and it was evaluated using simulations. Contrary to these works, our mechanism relies on machine learning to predict the thread mapping strategy without simulations. Instead, we use hardware counters and software libraries to gather information about the platform and applications.

Machine Learning. In [6], authors proposed a ML-based compiler model that accurately predicts the best partitioning of data-parallel OpenCL tasks. Static analysis was used to extract code features from OpenCL programs. These features were used to feed a ML algorithm which was responsible for predicting the best task partitioning among GPUs and CPUs. In [13], the authors proposed a two-staged parallelization approach combining profiling-driven parallelism detection and ML-based mapping to generate OpenMP annotated parallel programs. In this method, first they used profiling to identify portions of code that can be parallelized. Afterwards, they applied a previously trained ML-based prediction mechanism to each parallel loop candidate in order to select a scheduling policy from the four options implemented by OpenMP (*cyclic*, *dynamic*, *guided* or *static*). In [14], the authors proposed a ML-based approach to do thread mapping on parallel applications developed with OpenMP. The proposed solution was capable of predicting the ideal number of threads and the scheduling policy for an application. This approach was compared with the default OpenMP runtime through experiments on a Cell platform. In contrast to those works, we target a different domain of applications, *i.e.*, STM applications. These applications can be more sensitive to thread mapping due to their complex memory access patterns and effects of the underlying STM system.

6 Conclusion

In this paper, we proposed a dynamic thread mapping approach based on Machine Learning for TM applications. We focused on TM applications composed of multiple execution phases with potentially different transactional behavior in each phase. We defined and implemented this mechanism in a state-of-art STM system, making it transparent to the user. To the best of our knowledge, our work is the first to implement dynamic thread mapping for TM applications.

Our results showed that there is not a single thread mapping strategy adapted for all those complex applications. Instead, we could deliver a solution capable of detecting phase changes during the execution of the applications and then predicting a suitable thread mapping strategy adapted for each phase. We achieved performance improvements up to 31% in comparison to the best single strategy.

As future work, we aim at extending our predictor to consider a broader range of STM conflict detection and resolution policies. Additionally, we intend to consider more orthogonal TM characteristics to build even more diverse applications. Consequently, we can extend the evaluation of our approach over more diverse scenarios. Finally, we plan to use other machine learning algorithms to build new thread mapping predictors and compare their performances.

References

1. Broquedis, F., Clet-Ortega, J., Moreaud, S., Goglin, B., Mercier, G., Thibault, S.: hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In: PDP, pp. 180–186. IEEE Computer Society, Pisa (2010)
2. Castro, M., Góes, L.F.W., Ribeiro, C.P., Cole, M., Cintra, M., Méhaut, J.F.: A Machine Learning-Based Approach for Thread Mapping on Transactional Memory Applications. In: HiPC. IEEE Computer Society, Bangalore (2011)
3. Castro, M., Georgiev, K., Marangonzova-Martin, V., Méhaut, J.F., Fernandes, L.G., Santana, M.: Analysis and Tracing of Applications Based on Software Transactional Memory on Multicore Architectures. In: PDP, pp. 199–206. IEEE Computer Society, Aya Napa (2011)
4. Diener, M., Madruga, F., Rodrigues, E., Alves, M., Schneider, J., Navaux, P., Heiss, H.U.: Evaluating Thread Placement Based on Memory Access Patterns for Multi-core Processors. In: HPCC, pp. 491–496. IEEE Computer Society, Melbourne (2010)
5. Felber, P., Fetzer, C., Riegel, T.: Dynamic Performance Tuning of Word-Based Software Transactional Memory. In: PPoPP, pp. 237–246. ACM, NY (2008)
6. Grewe, D., O'Boyle, M.F.P.: A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 286–305. Springer, Heidelberg (2011)
7. Hong, S., Narayanan, S.H.K., Kandemir, M., Özturk, O.: Process Variation Aware Thread Mapping for Chip Multiprocessors. In: DATE, pp. 821–826. IEEE Computer Society, Nice (2009)
8. Hong, S., Oguntebi, T., Casper, J., Bronson, N., Kozyrakis, C., Olukotun, K.: Eigenbench: A Simple Exploration Tool for Orthogonal TM Characteristics. In: IISWC, pp. 1–11. IEEE Computer Society, Atlanta (2010)
9. Larus, J., Rajwar, R.: Transactional Memory. Morgan & Claypool (2006)
10. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: IISWC, pp. 35–46. IEEE Computer Society, Seattle (2008)
11. Quinlan, J.R.: Induction of Decision Trees. Machine Learning 1, 81–106 (1986)
12. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting Performance Data with PAPI-C. In: Parallel Tools Workshop, pp. 157–173. Springer, Berlin (2010)
13. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.: Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping. ACM SIGPLAN Not. 44, 177–187 (2009)
14. Wang, Z., O'Boyle, M.F.: Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. ACM SIGPLAN Not. 44, 75–84 (2009)
15. Zhang, J., Zhai, J., Chen, W., Zheng, W.: Process Mapping for MPI Collective Communications. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 81–92. Springer, Heidelberg (2009)