A Machine Learning-Based Approach for Thread Mapping on Transactional Memory Applications

Márcio Castro†, Luís Fabrício Wanderley Góes‡, Christiane Pousa Ribeiro*, Murray Cole†, Marcelo Cintra† and Jean-François Méhaut*

*INRIA - LIG Laboratory - Grenoble University, France
mail: marcio.castro@imag.fr, christiane.pousa@imag.fr, mehaut@imag.fr
†School of Informatics - ICSA - CARD Group - University of Edinburgh, UK
mail: lfwgoes.inf@ed.ac.uk, mic@staffmail.ed.ac.uk, mc@staffmail.ed.ac.uk

Abstract—Thread mapping has been extensively used as a technique to efficiently exploit memory hierarchy on modern chip-multiprocessors. It places threads on cores in order to amortize memory latency and/or to reduce memory contention. However, efficient thread mapping relies upon matching application behavior with system characteristics. Particularly, Software Transactional Memory (STM) applications introduce another dimension due to its runtime system support. Existing STM systems implement several conflict detection and resolution mechanisms, which leads STM applications to behave differently for each combination of these mechanisms. In this paper we propose a machine learning-based approach to automatically infer a suitable thread mapping strategy for transactional memory applications. First, we profile several STM applications from the STAMP benchmark suite considering application, STM system and platform features to build a set of input instances. Then, such data feeds a machine learning algorithm, which produces a decision tree to predict the most suitable thread mapping strategy for new unobserved instances. Results show that our approach improves performance up to 18.46% compared to the worst case and up to 6.37% over the Linux default thread mapping strategy.

Keywords—machine learning; software transactional memory; thread mapping.

I. INTRODUCTION

Multi-core processors are now a mainstream approach to deliver higher performance to parallel applications [1]. These platforms usually feature complex memory hierarchies composed of different levels of cache to reduce the penalties of accessing the main memory. From the application perspective, these memories can be viewed as low latency communication channels that allow threads to efficiently share data. However, these concurrent threads rely on memory affinity techniques to achieve high performance.

A well-known memory affinity technique is thread mapping [2], [3]. It places threads to specific cores to reduce memory latency or alleviate memory contention. For instance, threads that communicate often could be placed on cores that share some level of cache memory to avoid high latency accesses to the main memory. Thread mapping strategies can use prior knowledge of the application behavior [4] and/or the platform memory topology to map threads to cores. Although several mapping strategies have been proposed, there is no single solution that delivers high performance across different applications and platforms. Thus, the prediction of a suitable thread mapping strategy for a specific application/platform becomes a crucial problem.

Furthermore, Software Transactional Memory (STM) adds extra complexity to this decision process due to its runtime system support. The TM programming model allows programmers to write parallel code as transactions, which are guaranteed to execute atomically and in isolation regardless of eventual data races [5], [6]. 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. STM libraries usually implement different mechanisms to detect and solve conflicts. As a consequence, it becomes much more complex to determine a suitable thread mapping strategy for an application since it can behave differently according to the conflict detection and resolution mechanisms. A desirable solution would be an approach that could automatically infer a suitable thread mapping strategy to be applied considering the application, STM system and platform characteristics.

In this paper we propose a machine learning-based approach to do efficient thread mapping in transactional memory applications. First, we show that there is not a single thread mapping strategy that delivers high performance for STM applications. Second, we describe our machine learning-based approach to identify a suitable thread mapping strategy considering features of the application, STM and platform. Then, we evaluate the performance of our approach compared to individual thread mapping strategies and an oracle.

The rest of this paper is organized as follows. Section II presents the motivation for this research. Section III describes our machine learning-based approach to thread mapping. Section IV outlines our experimental setup while Section V presents results. Finally, Section VI discusses related works and Section VII concludes the paper.
Table I
IMPACT OF THREAD MAPPING STRATEGIES ON STM APPLICATIONS.

<table>
<thead>
<tr>
<th>Application</th>
<th>TinySTM</th>
<th>SwissTM</th>
<th>TL2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>var (%)</td>
<td>best</td>
<td>worst</td>
</tr>
<tr>
<td>Bayes</td>
<td>17.6</td>
<td>Round-Robin</td>
<td>Linux</td>
</tr>
<tr>
<td>Genome</td>
<td>26.8</td>
<td>Round-Robin</td>
<td>Compact</td>
</tr>
<tr>
<td>Intruder</td>
<td>14.0</td>
<td>Compact</td>
<td>Scatter</td>
</tr>
<tr>
<td>Kmeans</td>
<td>10.3</td>
<td>Linux</td>
<td>Compact</td>
</tr>
<tr>
<td>Labyrinth</td>
<td>9.7</td>
<td>Scatter</td>
<td>Compact</td>
</tr>
<tr>
<td>Soc2</td>
<td>25.9</td>
<td>Scatter</td>
<td>Compact</td>
</tr>
<tr>
<td>Vacation</td>
<td>9.2</td>
<td>Scatter</td>
<td>Compact</td>
</tr>
<tr>
<td>Yada</td>
<td>23.0</td>
<td>Compact</td>
<td>Scatter</td>
</tr>
</tbody>
</table>

II. MOTIVATION

On STM applications, the selection of a suitable thread mapping strategy for a specific application/STM/platform configuration is hard and there is no single solution. In this section, we show this fact by performing experiments with STM applications available from the Stanford Transactional Applications for Multi-Processing (STAMP) [7]. We ran all applications using 8 threads for three different STM systems (TL2 [8], TinySTM [9] and SwissTM [10]) and four thread mapping strategies (Round-Robin, Scatter, Compact and Linux). These strategies are presented in Figure 1. 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. This reduces memory latency access by sharing all levels of the cache hierarchy between concurrent threads. The Round-Robin strategy is an intermediate solution in which threads share higher levels of cache (e.g., L3) but not the lower ones (e.g., L2). Finally, the Linux default scheduling strategy is a dynamic priority-based one that allows threads to migrate to idle cores to balance the run queues.

As shown in Table I, performance variation can be high depending on the application and STM system.\(^1\) For instance, if the default Linux strategy is used for Bayes running on TinySTM, the performance is decreased by 17.6\% in comparison to the Round-Robin thread mapping strategy.

We also observe that the behavior of the same application varies across different STM systems. This stems from the fact that each STM system implements different mechanisms to detect and solve conflicts between transactions. Those mechanisms modify the behavior of the application, so the best thread mapping strategy is also affected. As we explain in Section IV-C, we focus on TinySTM as it implements a broad range of these mechanisms. Additionally, we observe similar variations within the same STM system for its different possible configurations.

As a conclusion, it is a hard problem to determine a suitable thread mapping strategy for a STM application considering both STM system and platform characteristics. The dynamic behavior of the STM system makes it even more non-trivial, since the same application can behave differently for each STM configuration. In the next section, we describe our solution to tackle this problem. We propose a machine learning-based approach to predict thread mapping strategies based on the STM system, application and platform characteristics.

III. APPLYING MACHINE LEARNING TO THREAD MAPPING

Machine Learning (ML) [11], [12], [13] has become a common component of approaches to model the behavior of complex interactions between applications, systems and platforms. It provides a portable solution to predict the behavior of new combinations of application/system, also called instances, based on a priori profiled runs. ML-based approaches share a common framework that is usually composed of a static and a dynamic phase. The static phase is subdivided in the following three major steps: i) application profiling; ii) data pre-processing and feature selection; and iii) learning process. Its target is to build up a predictor to be

\(^1\)The results for Bayes on TL2 are not available due to incompatibilities between the benchmark code and the TL2 library.
integrated in the actual system. In the dynamic phase, each new instance is profiled at runtime and then the collected data is used as an input to the predictor. Finally, the predictor outputs the target variable and the system applies it. Our proposed approach follows this methodology. In the rest of this section, we explain in detail each step of our ML-based approach.

A. Overview of the ML-Based Approach

Figure 2 presents a flow diagram describing each step of our approach. First, each instance of application/STM system has to be run and profiled. This profiling step involves gathering data of several characteristics, also known as features, available in the STM system and platform such as abort ratio, cache misses etc. Second, a database of profiled instances is built up and pre-processed in order to remove duplicated information, to normalize or to convert values, to extract new features and to remove features. In our approach, groups of instances that share the same feature values are merged to a single input instance sample where the target variable is the most suitable thread mapping strategy based on a ranking.

![Figure 2. Overview of the ML-based approach (static and dynamic phases).](image)

Then an ID3 decision tree algorithm [14] is fed with these input instance samples and trained. This training step finishes when the ID3 algorithm outputs a decision tree capable of predicting the most suitable thread mapping strategy for all the instances in the input database. To verify the decision tree accuracy, we apply a cross-validation technique. Finally, the predictor is integrated to the actual system so it can dynamically predict and apply a thread mapping strategy for each new unobserved instance.

B. Application Profiling

In the first step of the static phase, we have to select which features are relevant to be profiled. Table II presents the selected features classified into three categories as well as the target variable.

Features from Category A summarize the interaction between the application and the STM system. These selected features are commonly available across different STM implementations. Both transaction time ratio and abort ratio can be extracted and computed from execution time and STM statistics.

STM systems implement conflict detection and resolution mechanisms to handle concurrent transactional executions. A conflict can be detected as soon as it happens or postponed until a transaction commits. In the first mechanism, called eager conflict detection, as soon as a transaction access a shared data, all the other transactions become aware of this operation. In contrast, a STM system based on a lazy conflict detection mechanism only checks if there are any conflicts at commit time. This reduces the overhead during the transaction execution at the cost of increasing the overhead on a commit operation.

As soon as a conflict is detected, a conflict resolution mechanism is applied. Two common alternatives are to squash one of the conflicting transactions immediately (i.e., suicide strategy) or wait for an time interval before restart a conflicting transaction (i.e., backoff strategy). Category B shows the available mechanisms for both conflict detection and resolution on TinySTM.

In Category C, we observe the Last Level Cache (LLC) miss ratio to represent the interaction between the application and the platform. In order to measure the LLC miss ratio, it is necessary to access hardware counters. Tools such as the Performance Application Programming Interface (PAPI) [15] provide an interface to access such information.

Last, Table II presents the thread mapping strategies for our target variable (T) which our ML-based approach aims to predict. The selected strategies are the following ones as explained in Section II: Linux (LX), Compact (CP), Scatter (SC) and Round-Robin (RR).

C. Data Pre-Processing

The accuracy of a ML prediction model relies upon the quality of the input data samples. Thus, raw profiled data has to be pre-processed before feeding a ML algorithm. Our first step is to normalize or convert features according to the selected ML algorithm. As we explain in the next section, our selected ML learning algorithm works exclusively with categorical or discrete features. Since conflict detection and resolution mechanisms are already discrete, they do not need to be converted. For the other features, we apply a simple discretization technique to convert each ratio value within a range \((x; y)\) into one of the three following discrete
values: i) low \((0.0; 0.33)\); ii) medium \((0.33; 0.66)\); and iii) high \((0.66; 1.0)\). In order to reduce ambiguity among input data samples, it is still possible to increase the number of features and ranges. Although there exists more complex discretization techniques, we show in the results section that the selected features and ranges are enough to avoid biased input and consequently under-generalization of the resulting prediction model.

After data conversion, the second step is to determine the target variable value. In our approach, we aim to predict the most efficient thread mapping strategy for an application/STM/platform configuration. For this purpose, for each group of input instances that shares the same feature values (i.e., same application/STM system configuration) but with different thread mapping strategies, we select the most suitable strategy based on the application execution time. Then, each group of instances are merged and becomes a new single input instance in which the target variable is the mapping strategy that presented the lowest execution time.

Frequently, the execution time of different mapping strategies for the same application and STM configuration can be statistically equivalent. In that case, the decision of which strategy is the most suitable becomes non-trivial. From the ML algorithm perspective, ambiguous input instances should be avoided. For example, if strategy A is 0.1% faster than strategy B in one case and the opposite in another similar case, we should decide to elect just one strategy as the “fastest one” in both cases. Otherwise, the ML algorithm is not able to identify a common pattern on instances with similar behavior. Although the instances share similar feature values, they use different thread mapping strategies.

In order to address this ambiguity in our data samples, we propose an algorithm to rank the strategies before resolving these ties. This algorithm introduces an error percentage to distinguish mapping strategies that achieve very similar performance. First, for each group of instances that shares the same feature values except the target variable value, a mapping strategy can only be claimed as the best one, if its execution time is at least \(\varepsilon\) less than all the other ones. Second, for each thread mapping strategy, our algorithm counts how many times it was classified as the best one. This provides a ranking to distinguish all the other instances within a group that presented statistically similar performance. Thus, thread mapping strategies with higher counts have higher priority. Finally, we resolve all these ties by assigning to the target variable within a group the most suitable thread mapping strategy based on that ranking. This avoids having very similar input instances present different outcomes. Once the input data samples are pre-processed, our approach moves onto the learning phase.

### D. Learning Process

Several machine learning algorithms are available, including Neural Networks [16], Support Vector Machines (SVMs) [17] and Decision Trees [18]. Typically, a machine learning algorithm takes as input a data set \(S\) of input instances containing values for each corresponding feature in \(F\), including the target variable. It applies an iterative process on \(S\) to adjust its internal modeling parameters until it fits to the input data. This process is also known as the training phase. Then, it outputs a predictor (e.g., a set of functions, rules etc.) in which given a new input instance it predicts the target variable.

The ID3 learning algorithm outputs a decision tree as a predictor based on categorical discrete input instances [14]. Each internal node of the tree represents an input feature. Edges connecting one internal node to its successors are labeled with one of its feature values. Successor nodes can be either a feature or a value of the target variable. If the node contains the latter, it means that it is a leaf node.

In order to predict the value of the target variable, the predictor traverses its tree, matching the input instance values for each feature with the ones in the tree until it reaches a leaf node. Then it outputs the value within the leaf node as the predicted value.

STM features such as conflict detection and resolution mechanisms are inherently discrete. On the other hand, application features are continuous numerical values but are more representative if grouped into discrete values. In fact, they can be easily converted to discrete values using simple discretization techniques as we showed in the previous section. For that reason, ID3 becomes a suitable learning algorithm to address our problem. Additionally, a decision tree representation is simple and can be automatically integrated to any system.

Given a set \(S\) of input instances and a set of \(F\) features, the ID3 learning process starts choosing the “best” feature to be in the root node. By best, it means the feature that splits \(S\) in more homogenous subsets, that is, subsets where the majority of input instances contains the same target variable value. Homogeneity is measured by Entropy defined in Equation 1 where \(p_i\) is the probability that a given instance is in a subset with the same target variable value.

\[
\text{Entropy}(S) = \sum p_i \log_2(p_i)
\]  

\((1)\)

Gain can be derived from Entropy by computing the subtraction between the entropy of \(S\) and the sum of the entropies of its subsets \(S_i\) as shown in Equation 2. The feature with highest Gain is chosen to be the root node.

\[
\text{Gain}(S, F) = \text{Entropy}(S) - \sum_{i \in F} \frac{|S_i|}{|S|} \times \text{Entropy}(S_i)
\]  

\((2)\)

This recursive process is repeated for each successor node until all leaf nodes contain only input instances with the same target variable value.

In order to verify the accuracy of the resulting decision
tree, we use the standard leave-one-out cross-validation method. For each input instance, the learning algorithm is trained with the remaining instances and tested with the unused instance.

E. Prediction

Once the ID3 algorithm outputs a predictor and its accuracy is verified, the predictor is then integrated to the actual system. In this dynamic phase, it is able to determine which is the most suitable thread mapping strategy to be applied to a new unobserved instance. It is important to notice that the learning phase is no longer required for any new input instance (i.e., application/STM configurations). Instead, runtime profiling is needed in order to gather application and STM system information. In this case, an application starts running with a default thread mapping scheduling and during a initial warm-up interval it is profiled. The profiled data is converted to conform with the decision tree input format. The decision tree is then fed with the new input instance information and traversed, outputting a thread mapping strategy. These overheads are negligible since most profiled features are available natively and traversing a decision tree is done in logarithmic time. Finally, the system is able to switch to the predicted strategy.

IV. EXPERIMENTAL SETUP

A. Multi-core Platforms

In order to conduct our experiments, we have selected two representative multi-core platforms. SMP-24: a multi-core platform based on four six-core Intel Xeon X7460. Each group of two cores shares a L2 cache (3MB) and each group of six cores shares a L3 cache (16MB). SMP-16: a multi-core platform based on four quad-core Intel Xeon E7320 with 2MB of L2 cache shared per pair of cores. Table III summarizes the hardware characteristics of these platforms.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>SMP-24</th>
<th>SMP-16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores</td>
<td>24</td>
<td>16</td>
</tr>
<tr>
<td>Number of sockets</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Clock (GHz)</td>
<td>2.66</td>
<td>2.13</td>
</tr>
<tr>
<td>Last level cache (MB)</td>
<td>16 (L3)</td>
<td>2 (L2)</td>
</tr>
<tr>
<td>DRAM capacity (GB)</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>

B. Machine Learning

In the learning process of our approach, we obtained the results using the Weka tool [19]. It implements several machine learning algorithms including ID3. Weka also provides a cross-validation method to verify the accuracy of the generated decision tree. Lastly, we remark that each data point regarding the ML solution was predicted based on a decision tree trained without the data point itself. This leave-one-out cross-validation shows that our ML-based solution is able to predict thread mapping strategies for new unobserved instances.

C. Software Transactional Memory

In Section II, we showed that the impact of different thread mapping strategies depends on the STM application and STM system specific. In order to gain a deeper insight on such impact, we consider only the TinySTM [9] as our target STM system in the results section. Since TinySTM can be configured with different conflict detection and resolution mechanisms, we can have configurations that are similar to other STM systems and even other different configurations.

D. Transactional Memory Applications

We considered all applications available from STAMP [7] and performed the application profiling and data pre-processing phases described in Section III. Table IV shows an example of results obtained after these phases for a specific STM configuration and platform. In short, we used the maximum input data sizes for all applications as suggested in [7]. In Kmeans and Vacation, we selected a low contention configuration to cover a wider range of features. The LLC miss ratio was calculated based on statistics collected with PAPI [15]. It provides a consistent platform and operating system independent access to CPU hardware performance counters. On the SMP-24 and SMP-16, we gathered these statistics from the caches L3 and L2 respectively.

<table>
<thead>
<tr>
<th>Application</th>
<th>Tx Time Ratio</th>
<th>Tx Abort Ratio</th>
<th>LLC Miss Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bayes</td>
<td>high</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Genome</td>
<td>high</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td>Intruder</td>
<td>medium</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Kmeans</td>
<td>low</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>Labyrinth</td>
<td>high</td>
<td>high</td>
<td>medium</td>
</tr>
<tr>
<td>Ssca2</td>
<td>low</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>Vacation</td>
<td>high</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td>Yada</td>
<td>high</td>
<td>medium</td>
<td>medium</td>
</tr>
</tbody>
</table>

V. EXPERIMENTAL RESULTS

This section is divided into three parts. First, we perform an overall analysis of the impact of thread mapping strategies on the performance of STM applications. Next, we apply our ML-based approach to construct decision trees which will be used to predict an appropriate thread mapping strategy considering the application, STM system and platform features. Finally, we compare the performance of each individual thread mapping strategy with the one predicted by our ML-based approach and an oracle.
A. Overall Impact of Thread Mapping Strategies

We carried out experiments with all STAMP applications over the two platforms presented in Section IV. For each application, we varied the thread mapping strategy (Linux, Scatter, Compact and Round-Robin) as well as the conflict detection (eager or lazy) and resolution mechanisms (backoff or suicide) of the TinySTM system. Figure 3 shows the speedups of all possible combinations. The speedups refer to the parallel version of the applications with 8 threads in comparison to the corresponding sequential version. The results represent the averages obtained over a minimum of 20 executions to compute 95% confidence intervals.

The reason for presenting the results with 8 threads is twofold: (i) thread mapping significantly affects the performance when running a small number of threads due to cache sharing effects; and (ii) as the number of threads gets closer to the maximum number of available cores, thread mapping strategies converge to the same behavior: since STM applications tend to communicate uniformly (i.e., threads usually perform the same tasks), placing them on different cores does not impact the overall performance. Additionally, most of the applications do not scale beyond 8 threads.

As it can be observed, the SMP-24 is more sensitive to thread mapping than the SMP-16. Most of the applications presented a more significant variance in terms of speedup when different thread mapping strategies were applied. Specifically, a significant impact on the performance can be noticed on Labyrinth and Ssca2 on the SMP-24 whereas in SMP-16 such impact is much less perceivable. This is explained by the fact that SMP-24 has a more complex memory hierarchy (two shared cache levels). This leads to more complex performance tradeoffs in terms of memory contention and latency if threads are not mapped appropriately.

We also noticed that some applications are insensitive to the STM system parameters on both platforms. Modifying

---

Figure 3. Impact of thread mapping strategies on the performance of STM applications with different STM configurations.
the conflict detection and resolution mechanisms does not change the performance perceived by these applications. This can be explained by the fact that applications such as Sca2 do not spend much time within transactions.

In contrast, other applications such as Bayes, Genome and Labyrinth proved to be very sensitive to both thread mapping strategies and STM system parameters. On these applications, Compact usually delivered less performance gains than other thread mapping strategies.

This occurs for two reasons. From one perspective, Bayes and Labyrinth present high abort ratios. This means that transactions usually abort several times before committing. Each time a transaction is aborted, all the accesses to the shared data inside the transaction are re-executed. This fact increases the probability of having many transactions being executed at the same time, increasing the contention on cache memories. Since Compact forces threads to share cache, such contention is even higher. On the other hand, Genome is characterized by having a high transaction time ratio. This means that this application executes transactions most of the time. Since transactions frequently access shared data, they will dispute the access to shared caches. Thus, memory contention will be also increased.

### B. Generating the Decision Tree

As observed in the previous section, the target platform can influence the impact of the thread mapping strategy perceived by an application/STM configuration. Because of this, we trained our ML-based approach to predict a suitable thread mapping strategy considering each platform separately. The profiled data was obtained from runs with 8 threads for all possible feature combinations to feed the ML ID3 algorithm. By using 8 threads, the thread mapping strategies can explore all levels of cache memory and the resulted decision trees are still valid for less threads. As aforementioned in Section V-A, when the number of threads gets close to the maximum number of available cores, the thread mapping strategy does not impact the performance.

Figure 4 shows two decision trees generated by the ID3 learning algorithm on the SMP-24 (Figure 4a) and SMP-16 (Figure 4b). It is important to mention that, due to the ranking algorithm in the pre-processing step, some thread mapping strategies can be dropped from the decision tree even when they are actually the best one for a given instance. This occurs when the thread mapping strategy does not present significantly more performance gains than other strategies. That was the case of the Linux strategy in the decision tree for SMP-24.

The influence of the target platform is confirmed by the decision trees, which are very different from each other. On the SMP-24, Compact and Scatter are the two most important thread mapping strategies whereas Round-Robin and Compact are the most important ones on the SMP-16. We can also observe that the decision tree of the SMP-16 is much simpler than the SMP-24. This can be explained by the fact that the SMP-24 has a more complex cache hierarchy.

Even though there are differences between the decision trees, we can derive some overall conclusions. First, there is a possible correlation between the abort ratio and the LLC miss ratio. On both trees, when an application presents a high abort ratio, the LLC miss ratio is taken into account to decide the thread mapping policy to be applied. Second, when the abort ratio is low, the predictor tends to select a strategy that places threads far from each other (e.g., Scatter and Round-Robin strategies). A low abort ratio means that transactions rarely access the same shared data at the same time. Thus, the contention generated by several threads accessing the same cache can be alleviated by applying such strategies.

Finally, when the conflict detection used is backoff, the decision tree tends to decide for the Compact thread mapping strategy. Backoff forces transactions to wait some time before re-executing due to aborts. This reduces the amount of contention on the cache, making it possible to place threads on sibling cores to amortize the access latency.

An exception is the Linux default strategy that appears only in one specific case. It tends to distribute and migrate threads among cores in order to balance the workload. Thus, the resulting distribution of threads is variable and unpredictable. These characteristics thus benefit applications...
with low cache miss ratio and very dynamic behavior. This is the case of *Bayes* on SMP-16 in which the lazy detection mechanism contributes to reduce the number of false aborts compared to the eager mechanism.

Finally, the cross-validation method allows to quantify the accuracy of a decision tree or any other predictor. The accuracy is calculated dividing the number of correctly predicted instances by the total number of instances. Respectively, the accuracies on the SMP-24 and SMP-16 platforms were 86% and 72%. In the next section we show that although the accuracy is not higher than 90%, the average performance achieved is comparable to the oracle.

**C. Predicting the Thread Mapping Strategy**

Figure 5 shows the average speedup over all benchmarks considering the fixed thread mapping strategies, our ML approach and the oracle. Particularly, the speedups of the four STM configurations (eager/lazy X suicide/backoff) for a specific thread mapping strategy, as shown in Figure 3, are averaged and reduced to a single bar. The main idea is to show the performance of each application and thread mapping strategy considering all STM configurations at the same time. In the last column, it also presents the overall average speedup for each strategy.

The oracle is an upper bound strategy that always choose the best thread mapping strategy for each application/STM/platform configuration. As a result, the oracle can achieve average speedup higher than the best fixed mapping strategy due to application performance variations across multiple STM configurations. In the ML-based version, for each configuration, the prediction is based on a decision tree trained *without* that specific configuration.

Overall, on the SMP-16 and SMP-24 platforms the ML-based approach presented average performance improvements of 18.46% and 11.35% respectively compared to the worst case and 6.37% and 3.21% over the Linux default strategy. On both cases, the ML-based approach is within 1% of the oracle performance.

In *Genome* on SMP-16, the *Compact* strategy leads to 32% less performance than the oracle. This confirms that choosing the wrong thread mapping strategy can lead to high performance loss. On the other hand, *Compact* has 18% better performance on average compared to the other fixed strategies on *Yada* for the same platform. This suggests that a flexible approach to select the most suitable thread mapping strategy is required. In fact, the ML-based approach is equal to the oracle on *Yada*.

In cases in which a single thread mapping strategy is sensitive to application/STM features, the ML solution achieves much higher average speedup compared to these single strategies. This stems from the fact that the machine learning approach considers the STM system and application features to choose a thread mapping strategy. This reduces the variability and leads to higher speedups, as we observe in *Bayes*.
VI. RELATED WORK

Thread Mapping. Some studies on thread mapping for MPI and OpenMP applications are relevant to our current work, since they propose heuristics to map threads on multi-core platforms. In [20], the authors presented a process mapping strategy for MPI applications with collective communications. The strategy uses the graph partitioning algorithm to generate an appropriate process mapping for an application. The proposed strategy was then compared with Compact, Scatter and the solution presented in [21].

In [4], two thread mapping algorithms are proposed for applications based on the shared memory programming model. These algorithms rely on memory traces extracted from benchmarks to find data sharing patterns between threads. During the profiling phase, these patterns are extracted by using Simics [22] simulator, running the workloads in a simulated UltraSPARC machine. The proposed approach was compared to Compact, Scatter and the operating system process mapping strategies.

In [2], the authors proposed a dynamic thread mapping strategy for regular data parallel applications implemented with OpenMP. The strategy considers the machine description and the application characteristics to map threads to processors. The efficiency of the proposed thread mapping strategy was evaluated using the Simics [22] simulator.

In contrast to these works, we propose a ML-based approach for deciding an appropriate thread mapping strategy considering both application and platform features. We do not rely on simulation to gather information about the application, STM system and platform features during the profiling phase.

Machine Learning. In [23], authors proposed a ML-based compiler model that accurately predicts the best partitioning of data-parallel OpenCL tasks. Static analysis is used to extract code features from OpenCL programs. These features are used to feed a ML algorithm which is responsible for predicting the best task partitioning. The prediction is composed by two stages. In the first stage, the model predicts whether tasks should be only mapped to GPUs or CPUs. If the first stage of prediction does not lead to a conclusion, the program is then passed to the second stage which is responsible for mapping tasks to both GPUs and CPUs. The number of tasks to be mapped to GPUs and CPUs is determined by the predictor.

In [24], 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 use profiling data to extract control and data dependencies to identify portions of code that can be parallelized. Afterwards, they apply 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 [25], the authors proposed a ML approach to thread mapping on parallel applications developed with OpenMP. Using the machine learning approach, the proposed solution is capable of predicting the number of threads and the scheduling policy for an application. In order to evaluate the performance of the proposed solution, authors compared their approach to the default OpenMP runtime on a Cell platform.

In contrast, we focus on STM applications, which can be more sensitive to thread mapping due to their complex memory access patterns. Furthermore, since STM applications have a more dynamic behavior, our ML approach also considers the STM system features.

Performance Evaluation of STM Applications. In [13] a study of the performance of several STM implementations on big multi-core machines is presented. Authors evaluated three STM implementations using the STM Bench7 [26] and the Simics simulator [22]. Results showed that depending on the design choice of a STM system, the performance of STM applications can vary significantly. Although they evaluated the performance of different STM implementations on multi-core architectures, authors focused their study in the influence of the memory access latencies in the STM performance. Indeed, they did not considered the impact of thread mapping in their evaluation.

In [27], authors designed and implemented a scheduling support mechanism for STM applications in the Linux and Solaris kernels. Such support allows the operating system to cooperate in order to reduce the number of aborts and increase the transaction throughput. Although this approach presents significant improvement gains, it demands modifications in the kernel of the operating system.

In contrast, we are interested on the impact of thread mapping strategies on the performance of STM applications. Our approach does not require any changes in the operating system, application or STM system. Moreover, the choice of the thread mapping to be applied does not rely on the user’s knowledge of the system. Our approach is also portable across different operating systems and platforms.

VII. CONCLUSIONS

Predicting a suitable thread mapping strategy to be applied to STM applications is not a trivial task. Due to several conflict detection and resolution mechanisms implemented by STM systems, STM applications can behave very differently in the same platform. Additionally, the target platform features have an important influence on the right choice of a thread mapping strategy.

In this paper we proposed a ML-based approach to automatically choose an appropriate thread mapping strategy for STM applications considering the features of the applications, STM system and platform. Our solution is portable
across different platforms and can be easily extended to consider other features such as new conflict detection and resolution mechanisms, to cover a wider range of STM system features.

Our ML-based approach is composed of four main phases. In our results, we initially profiled all applications available from STAMP benchmarks to gather information about their features. For each run, we varied the STM system features (conflict detection and resolution mechanisms) and thread mapping strategies. These experiments were carried out on two multi-core platforms. The profiled information was then pre-processed to adapt it to the ML algorithm. After that, the pre-processed data was used to feed an ID3 algorithm. As a result, the algorithm produced a decision tree which is used to predict a suitable thread mapping strategy to be applied to other unobserved instances.

Results obtained in this work showed that our ML-based approach can improve the performance of STM applications by up to 18.46% compared to the worst case and up to 6.37% over the Linux default thread mapping strategy. We then applied a leave-one-out cross-validation to estimate how accurately our predictive model will perform over future as yet-unseen instances. We achieved accuracies of 86% and 72% for the SMP-24 and SMP-16 platforms respectively. Although the accuracies were not higher than 90%, the average performance achieved was within 1% of the oracle performance.

As future works, we intend to apply this approach to a broader range of STM conflict detection and resolution mechanisms and other features. It is also desirable to integrate this approach to be automatically executed in an existing STM system. Finally, we also intend to compare this ML-approach with fully dynamic approaches.

REFERENCES


