Autonomic Parallelism and Thread Mapping Control on Software Transactional Memory

Naweiluo Zhou, Gwenaël Delaval, Bogdan Robu, Éric Rutten, Jean-François Méhaut

Autonomic Parallelism and Thread Mapping Control on Software Transactional Memory

Naweiluo Zhou[1], Gwenaël Delaval[1] , Bogdan Robu[2]
Univ. Grenoble Alpes, LIG, CNRS, INRIA[1]
Univ. Grenoble Alpes, GiPSA-Lab, CNRS[2]
Grenoble, France
naweiluo.zhou@inria.fr, gwenael.delaval@inria.fr,
bogdan.robu@gipsa-lab.grenoble-inp.fr

Éric Rutten, Jean-François Méhaut
Univ. Grenoble Alpes, LIG, CNRS, INRIA
Grenoble, France
eric.rutten@inria.fr, jean-francois.mehaut@imag.fr

***Abstract*—Parallel programs need to manage the trade-off between the time spent in synchronization and computation. The time trade-off is affected by the number of active threads significantly. High parallelism may decrease computing time while increase synchronization cost. Furthermore thread locality on different cores may impact on program performance too, as the memory access time can vary from one core to another due to the complexity of the underlying memory architecture. Therefore the performance of a program can be improved by adjusting the number of active threads as well as the mapping of its threads to physical cores. However, there is no universal rule to decide the parallelism and the thread locality for a program from an offline view. Furthermore, an offline tuning is error-prone. In this paper, we dynamically manage parallelism and thread localities. We address multiple threads problems via Software Transactional Memory (STM). STM has emerged as a promising technique, which bypasses locks, to address synchronization issues through transactions. Autonomic computing offers designers a framework of methods and techniques to build autonomic systems with well-mastered behaviours. Its key idea is to implement feedback control loops to design safe, efficient and predictable controllers, which enable monitoring and adjusting controlled systems dynamically while keeping overhead low. We propose to design a feedback control loop to automate thread management at runtime and diminish program execution time.**

***Keywords*—autonomic, transactional memory, feedback control, synchronization, parallelism adaptation, thread affinity**

I. Introduction

Multicore processors are ubiquitous, which enhance program performance through thread parallelism (number of simultaneous active threads). The complexity of memory hierarchy brought by multicore processors gives diverse access latency or memory contention from threads located on different cores. Allocating threads appropriately to certain cores can potentially improve usage of resource, such as main memory, cache and interconnections. Therefore the overall performance of a parallel application not only depends on the parallelism degree but also the locality of its threads. High parallelism shortens execution time, but it may also potentially increase synchronization time. The execution time can be further deteriorated with an unsuitable placement of the active threads on the cores. The conventional way to address multiple-thread issues is via locks. However, locks are notorious for various issues such as the likelihood of deadlock and the vulnerability to failure and faults. Also it is not straightforward to analyse interactions among concurrent operations.

Transactional memory (TM) emerges as an alternative synchronization technique, which addresses synchronization issues through transactions. Accesses to shared data are enclosed in transactions which are executed speculatively without being blocked by locks. Various TM schemes have been developed [1], [2], [3] including Hardware Transactional Memory (HTM), Software Transactional Memory (STM) and Hybrid Transactional Memory (HyTM). In this paper, we present runtime thread control under STM systems where synchronization time originates in transaction aborts. There are different ways to reduce aborts, such as the design of contention manager policy, the way to detect conflicts, the setting of version management and the parallelism degree.

Online thread parallelism adaptation began to receive attention recently. A suitable parallelism degree in a program can significantly affect its performance. However it is onerous to set a suitable parallelism degree for a program offline especially for the one with online behaviour variation. When apropos of the program with online behaviour fluctuation, there is no unique parallelism that can enable the optimal performance. Therefore the natural solution consists in monitoring the program at runtime and alter its parallelism degree when necessary. Additionally, the complexity of memory hierarchy imposes differences of memory access time among cores, which consequently lead to the performance diversity related to the diverse locations of threads. When the number of active thread number varies, their corresponding locations may also need to be adjusted accordingly in order to optimize usage of memory hierarchy. Assigning threads to specific cores is called thread mapping [4] and fixing a thread to a specific core is called setting the thread affinity.

We introduce feedback control loops to STM systems to achieve autonomic computing [5], *i.e.* to automatically regulate thread number and their mapping to cores at runtime. In this paper we argue that online thread management is necessary and feasible in STM systems. We demonstrate that the program performance is sensitive to the parallelism degree as well as thread mapping. We present one effective profiling and control framework for thread management using TinySTM [2].

The main contributions of our paper are as follows:

1) We build a runtime parallelism predictor based on probability theory.
2) We utilise a feedback control loop to optimize both parallelism and thread mapping strategies at runtime for STM systems.

The rest of the paper is organized as follows. Section II summaries the background and related work. Section III details the profiling procedure and the design of feedback control loop that optimizes the parallelism and thread mapping strategy. Section IV presents the implementation details. Section V shows the results. Section VI discusses the pros and cons of our models and Section VII concludes the paper and gives future work.

II. Background and Related Work

A. Software Transactional Memory

Transactional memory (TM) is an alternative synchronization technique. In TM, data accesses to shared memory are enclosed in transactions which are executed speculatively without being blocked by locks. Each transaction makes a sequence of tentative changes to shared memory. When a transaction completes, it can either *commit* making the changes permanent to memory or *abort* discarding the previous changes made to memory [1]. Two parameters are often used in TM to indicate system performance, namely *commit ratio* and *throughput*. Commit ratio (CR) equals the number of commits divided by the sum of number of commits and aborts; it measures the level of conflicts or contention among current transactions. Throughput is the number of commits in one unit of time; it directly indicates progress of useful work. In our proposal we propose to use logic time to mark the profile period. This is due to the fact that various TM applications vary in the size of transaction leading to significant variation of execution time. TM can be implemented in software, hardware or hybrid. Different mechanisms explore the design trade-off that impacts on performance, programmability and flexibility. In this paper, we focus on STM systems and utilise TinySTM [2] as our experimental platform. TinySTM is a lightweight STM system that adopts a shared array of locks to control the concurrent accesses to memory and applies a shared counter as clock to manage its transaction conflicts.

B. Parallelism and Thread Mapping

Performance of STM systems has been continuously improved. Studies to improve STM systems mainly focus on the design of *conflict detection*, *version management* and *conflict resolution*. Conflict detection decides when to check read/write conflicts. Version management determines whether logging old data and writing new data to memory or vice versa. Conflict resolution, which is also known as contention management policy, handles the actions to be taken when a read/write conflict happens. The goal of the above designs is to reduce wasted work. The amount of wasted work resides in the number of aborts and the size (the number of operations inside an abort) of aborts. The higher contention in a program, the larger amount of wasted work. The time spent in wasted work is the synchronization time in the STM view. Apart from diminishing wasted work, one way to improve STM system performance is to trim computing time. High parallelism may accelerate computation but resulting in high contention thus high synchronization time. Hence parallelism can significantly affect performance of a program.

Modern computing systems carry a complex memory hierarchy that gives different access latency to main memory from different cores, thus the mapping of threads to the cores impact on the application's performance. *Castro* [4] defines four different thread mapping strategies based on the locations of threads, which are addressed as **Compact** (threads are physically placed on sibling cores, *e.g.*, on C0 and C1), **Scatter** (threads are distributed across processors, *e.g.*, on C0 and C4), **RoundRobin** (is an intermediate solution, threads share higher levels of cache (*e.g.*, L3) but not the lower ones (*e.g.*, L2).) and **Linux** (the default Linux thread scheduling strategy). Fig. 1 illustrates the four thread mapping strategies when concerning with two threads.

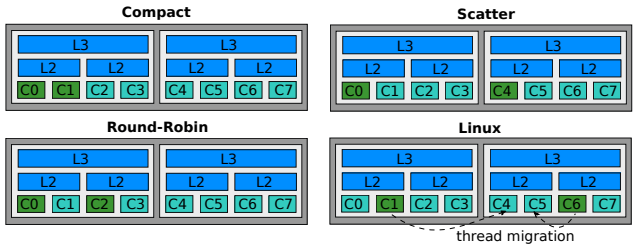


Fig. 1. Four thread mapping strategies [4] when concerning with two threads. The processor structure is only an illustration.

C. Autonomic Computing

Autonomic computing [6] is a concept that brings together many fields of computing with the purpose of creating self-managed computing systems.

In this paper, we introduce a feedback control loop to achieve autonomic thread control. A classic feedback control loop is illustrated in Fig. 2 in the shape of a MAPE-K loop.

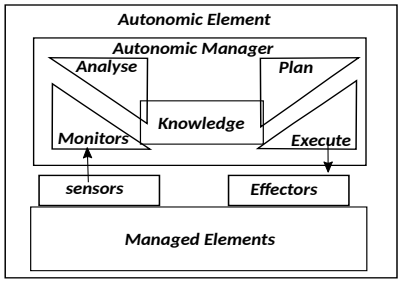


Fig. 2. A MAPE-K control loop. It incorporates an autonomic manager, sensors, effectors as well as managed elements among which the autonomic manager plays the main role.

In general, a feedback control loop is composed of (1) an autonomic manager, (2) sensors (collect information), (3) effectors (carry out changes), (4) managed elements (any software or hardware resource). An autonomic manager is composed of five elements: a monitor (used for sampling), an analyser (analyse data obtained from the monitor), knowledge (knowledge of the system), plan (utilise the knowledge of the system to carry out computation) and execute (perform

changes). It is worth noting that the above five elements of the autonomic manager can overlap with each other.

D. Related Work

It has been been addressed in a few previous work [7], [8], [9], [10] to dynamically adapt parallelism via control techniques to reduce wasted work. *Ansari et. al.* [7] proposed to adapt the parallelism online by detecting the CR changes of applications. The parallelism is regulated if the CR falls out of the preset CR range or is not equal to a single preset CR value. This is based on the fact that CR falls during highly contended phases and rises with low contended phases. *Ansari et. al.* gave five different algorithms that decide the profile length and the parallelism degree. *Ravichandran et. al.* [10] presented a model which adapts the thread number in two phases: *exponential* and *linear* with a feedback control loop. *Rughetti et. al.* [8] utilise a neural network to enable the performance prediction of STM applications. The neural network is trained to predict the wasted transaction execution time which in turn is utilised by a control algorithm to regulate parallelism. *Didona et. al.* [9] provides an approach to dynamically predict the parallelism based on the workload (duration and relative frequency, of read-only and update transactions, abort rate, average number of writes per transaction) and throughput, through one feedback control loop its prediction can be continuously corrected. There is only one previous work from *Castro* [4] that investigated the runtime thread mapping issues on STM applications. *Castro* utilises an offline training procedure to obtain a thread mapping strategy predictor and employs it to estimate a thread-to-core mapping solution at runtime. No prior literature has addressed the issue on coordination of parallelism and thread mapping for TM systems. *Wang et. al.* [11] present an offline compiler-based approach for OpenMP programs. Two machine learning algorithms (that require offline data training), namely *feed-forward Artificial Neural Network* (ANN) and *Support Vector Machine* (SVM) are employed, to dictate parallelism degree and thread mapping rules respectively.

Our approaches differ from the previous work as (1) comparing with *Ansari et. al.*, we resolve the CR range which is adaptive to the online program behaviour (2) no modifications are made to applications to perform thread management (3) analogising with the parallelism prediction by *Didona et. al.*, *Ravichandran et. al.* and *Rughetti et. al.*, we predict the optimal parallelism by a model based on probability theory that requires no offline training procedure or trying different number of threads to search the optimum value. (4) contrasting with the offline approach of *Wang et. al.* on OpenMP, we design a feedback control loop which regulates both parallelism and thread mapping strategy at runtime for STM systems.

III. Autonomic Thread Adaptation

In this section, we present the design of the feedback control loop for thread management. We firstly give an overview of the profiling algorithm and later detail the algorithm via the prism of control theory.

We measure three parameters from the STM system, namely the number of commits, the number of aborts and physical time. The number of commits and the number of aborts are subsequently addressed as commits and aborts. As stated in Section II-A, a *commit* denotes a transaction successfully accomplishes its operations and an *abort* means a transaction fails to finish its operations. We choose CR and throughput to indicate program performance, as CR and throughput are both sensitive to thread variation. Either is by itself not sufficient enough to represent program performance:

- A high throughput means fast program execution whereas a low throughput denotes slow program progress. But a low throughput may be caused by a low parallelism degree or simply just by a low number of transactions taking places.
- CR indicates the conflicts among threads. A high CR means low synchronization time whereas a low CR means high synchronization cost. But a low CR can bring a high throughput when a large number of transactions are executing concurrently, whereas a high CR may give low throughput due to a small number of transactions executing concurrently.

Therefore it is necessary to utilise both CR and throughput to indicate program performance. The controller observes CR to detect the contention fluctuation and enable the corresponding control actions. The control actions are verified by checking if the throughput is improved after the taken actions.

A. Overview of Profiling Algorithm

To achieve autonomic thread adaptation which enables a program to work under optimized parallelism and thread mapping strategy, we propose to periodically profile the applications at runtime. Parallelism and thread mapping strategies both impact on application performance. But the influence of parallelism to an application is much larger than the impacts of thread mapping strategy, as we can see later in Section V. Furthermore, a different parallelism degree may require a different thread mapping strategy [4] to enhance its performance but a different thread mapping strategy has low effect on estimation of parallelism degree. Hence we profile the thread mapping strategy based on the active number of threads chosen.

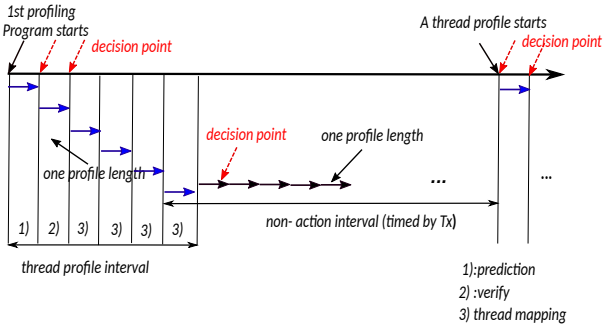


Fig. 3. Periodical profiling procedure. At each decision point (marked by dashed red arrow), the actions are taken. One profile length is a fixed number of commits.

By observing CR, we can obtain the contention information of an application. CR usually fluctuates in a certain range within the same phase. When a program enters a new phase, the current parallelism and thread mapping strategy produces

a different CR that falls out of the current CR range. The CR fluctuation triggers a new thread control action. Initially the two CR thresholds (CR_UP, CR_LOW) are both set to be 0 and are trained in the later profile stage. When the CR falls out of the thresholds, it may be required to add or remove threads to resolve conflicts in order to optimize the throughput. The throughput can be further enhanced by a good thread mapping strategy. The detail of the procedure is illustrated in Fig. 3.

The profiling procedure commences once the program starts with **Linux** as its initial thread mapping strategy, since **Linux** is the default thread mapping strategy on Linux systems. Initially the program creates a pool of threads, of which some can be activated and some can be suspended. At each *decision point*, the control loop (see section III-B) is activated to regulate the parallelism and thread mapping strategy or suspend this regulation. We address the two operations as *thread regulation*. As shown in Fig. 3, a *profile length* is a fixed length of logic time (commits) for information gathering, such as commit, abort and time. A *thread profile interval*, within which the thread regulation is enabled, is composed of a continuous sequence of profile lengths. The *non-action interval*, within which the thread regulation is suspended, is composed of one or a continuous sequence of profile lengths. The duration of the thread profile interval and non-action interval are not fixed values as shown in Fig. 3. The thread profile interval can be composed of two profile lengths (only parallelism regulation) or six profile lengths (thread regulation). When the thread number produced by the parallelism predictor remains unchanged or is over half of the core number, the thread profile interval terminates. Otherwise, thread mapping strategies are profiled. The reason that half of the core number is chosen as a critical point is: (1) when the parallelism reaches the maximum value, the thread mapping strategy gives little impact on program performance as all the possible cores are occupied; (2) the higher parallelism, the less impact received from different thread mapping strategies. A non-action interval terminates when CR falls out of the CR range. The above procedure continues until the program terminates. It is worth noting that a decision point in Fig. 3 corresponds to one state in Fig. 4(b) as explained later in Section III-B.

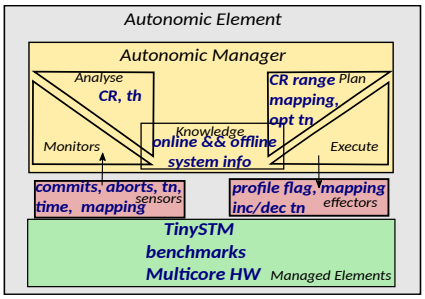
B. Feedback Control of Thread Adaptation

Fig. 4(a) gives the structure of the complete platform that forms a MAPE-K feedback control loop. The autonomic manager, which can be also seen as the controller, is described as an automaton in the paper as shown in Fig. 4(b). Our designed automaton is composed of five states, and the program can only reside on one state at each decision point.

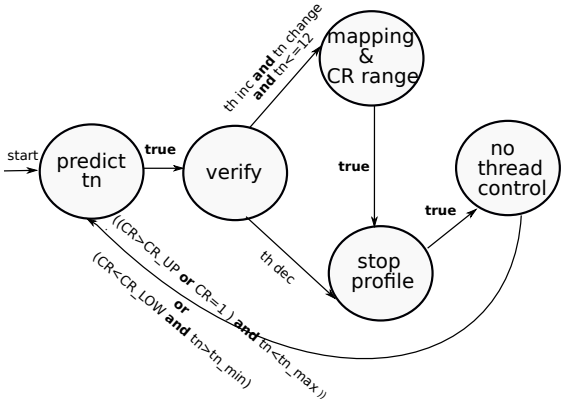
1) Control Objective: Under control theory terminology, the control objective of the feedback control loop is to maximize throughput and reduce the application execution time. This is achieved by minimizing the conflicts among threads and diminishing thread migration among cores.

2) Inputs and Outputs: The inputs of the loop are commits, aborts, active thread number, thread mapping strategy and physical time. The outputs/actions are optimum parallelism, optimum thread mapping strategy and profile action.

3) Four Decision Functions: The control loop is activated at each decision point (see Fig. 3). Four decision functions



(a) The instantiation of MAPE-K-shape feedback control loop.



(b) The structure for the autonomic manager of Fig.4(a) in automaton shape.

Fig. 4. The feedback control loop. th, tn and mapping stand for throughput, thread number and thread mapping strategy respectively. One state corresponds to one decision point in Fig. 3. The boolean value **true** means unconditional state transfer.

cooperate to make decisions: *a parallelism predictor* (predicts the parallelism degree), *a thread mapping strategy decision function* (predicts the thread mapping strategy), *a CR range decision function* (decides the phase change) and *a thread profile decision function* (enables the thread profile action). At each decision point as illustrated in Fig. 3, one corresponding decision function reacts to make its decision. We firstly detail the derivation of the parallelism predictor, then we describe the design of the automaton as it elucidates the relation among the decision functions, as well as how the thread mapping decision function and profile decision function are designed. The CR decision function is presented lastly.

We have developed a probabilistic model to serve as the parallelism predictor. This probabilistic model however has its limitation, it is based on two assumptions:

- we assume that the same amount of transactions are executed in the active threads during a fixed period, as every thread shows similar behaviour in our TM applications,
- the probability of one *commit* (see Section II-A for definition) approaches a constant, as there is a large amount of transactions executed during the fixed period making the probability of conflicts between two transactions approaches a constant.

Let L_0 be a fixed period during an application execution, assuming that the average length of transactions (including

the aborted transactions and the committed transactions), is L, thus the number of transactions N executed during L_0 can be expressed in equation (1).

$$N = n \cdot \frac{L_0}{L} = \alpha \cdot n \tag{1}$$

Where n stands for the number of active threads during L_0, N contains both aborts and commits, and $a = \frac{L_0}{L}$.

We assume that the probability of the conflicts p between two transactions is independent from the current active threads, thus independent from the number of active transactions. Therefore during the L_0 period, one transaction can commit if it encounters no conflicts with other active transactions. The probability of a commit can be expressed in Equation (2).

$$P(X_i = 1) = q^{(N-1)} \tag{2}$$

Where $q = 1-p$, which stands for the probability of a commit between two transactions. However, the transactions executed in a sequence within the same thread do not cause conflicts among each other, the probability of a commit in Equation (2) is lower than the reality. We suppose that during L_0 period, each thread approximately executes the same number of transactions which is $\frac{N}{n}$. Therefore the number of transactions causing conflict are reduced to $N - \frac{N}{n}$. So Equation (2) can be modified as in Equation (3).

$$P(X_i = 1) = q^{(N-\frac{N}{n})} \tag{3}$$

Equation 3 is correct only if there is a large amount of transactions executed during L_0 period making the probability of conflicts p between two transactions approaches a constant, thus q approaches a constant.

Under the terminology of probability theory, X_i is a random variable with $X_i = 1$ if the transaction i is committed, $X_i = 0$ if i aborted. X_i follows a Bernoulli law of parameter $q^{(N-\frac{N}{n})}$.

Let T represents the throughput. In a unit of time, the throughput can be also expressed as $T = \sum X_i$ and CR can be expressed as $CR = \frac{T}{N}$. As T is a random variable which follows a binomial distribution $B(N, q^{(N-\frac{N}{n})})$. The expected value of T is therefore to be:

$$E[T] = N \cdot q^{(N-\frac{N}{n})} = \alpha n q^{\alpha(n-1)} \tag{4}$$

Hence

$$E[CR] = q^{(N-\frac{N}{n})} = q^{\alpha(n-1)} \tag{5}$$

Equation (4) can be rewritten as a function from n to T as shown in Equation (6).

$$T(n) = \alpha \cdot n \cdot q^{\alpha(n-1)} \tag{6}$$

To obtain the value of n where the throughput can reach the maximum, we compute the derivative of Equation (6) as shown in Equation (7).

$$T'(n) = \alpha q^{\alpha(n-1)} + \alpha^2 n q^{\alpha(n-1)} \ln(q) \tag{7}$$

Therefore

$$\begin{aligned} T'(n_{opt}) = 0 &\Leftrightarrow \alpha q^{\alpha(n_{opt}-1)} + \alpha^2 n_{opt} q^{\alpha(n_{opt}-1)} \ln(q) = 0 \\ &\Leftrightarrow q^{\alpha(n_{opt}-1)} \cdot (\alpha + \alpha^2 n_{opt} \ln(q)) = 0 \\ &\Leftrightarrow \alpha + \alpha^2 n_{opt} \ln(q) = 0 \\ &\Leftrightarrow n_{opt} = -\frac{1}{\alpha \ln(q)} \end{aligned} \tag{8}$$

Where n_{opt} stands for the optimum value of n which is the optimum thread number.

From Equation (5), we can obtain $q = CR^{\frac{1}{\alpha(n-1)}}$. Then Equation (8) can be rewritten as follows.

$$n_{opt} = -\frac{n-1}{\ln(CR)} \tag{9}$$

Where n_{opt} stands for the optimum thread number, n stands for the number of current active threads and CR is the current commit ratio.

In this paragraph, we detail the thread mapping strategy decision function and the thread profile decision function. The automaton as illustrated in Fig. 4(b) starts from the *predict tn* state where the optimum parallelism is predicted. Executing the predicted parallelism for one profile length, the automaton unconditionally enters the *verify* state which corresponds to the second decision point in Fig. 3. In the *verify* state, the predicted optimum thread number is verified by comparing the current throughput with the previous throughput. If the current throughput is larger than the previous value, we maintain the predicted thread number, otherwise the thread number is switched back to the previous optimal value. Additionally the *verify* state decides the necessity of profiling the thread mapping strategy. The following conditions must be both satisfied in order to enter *mapping & CR range* state: (1) the thread number has been changed, (2) the new thread number is less than half of the core number (see the reasoning in Section III-A). If the conditions are not met, the automaton goes into *stop profile* state making all the thread regulation suspended. At each instant of *mapping & CR range* state, a new thread mapping strategy is applied. The optimum thread mapping strategy is the one which yields the highest throughput. *mapping & CR range* state is executed four times as four thread mapping strategy needs to be profiled. At the final decision point of a thread profile interval, the parallelism and thread mapping strategy are set to be the value which yields the maximum throughput. At each decision point of a non-action interval, the profile decision function decides if a new thread profile is needed, which corresponds to the *no thread control* state. A new thread profile procedure starts from the *predict tn* state. It is worth noting that the profile decision function is performed on the *verify*, *mapping & CR range* and *no thread control* state.

Lastly we describe the CR range decision function. Through detecting CR fluctuation in the state *no thread control*, the controller decides whether the program enters a new phase. When CR fluctuates in a certain range, there is no need to regulate the threads as the conflicts in the program are already minimized by previous control actions. However it is onerous to determine such a CR range offline, especially it is unrealistic to set a fixed CR range for the programs with online behaviour variation. Also a constant CR range impedes programs to search its optimum parallelism and thread mapping strategy.

Therefore it becomes interesting to dynamically resolve a CR range. The CR range computation is based on the optimum CR value. The optimum CR (CR_{opt}) is produced by the optimum parallelism and thread mapping strategy and has been recorded by the previous operations. The two thresholds of the CR range is the optimum CR plus or minus its 10% ($CR_{thresholds} = CR_{opt} \pm 0.1 * CR_{opt}$). This means that when CR fluctuates within 10% of the optimum CR, the program still stays in the same phase and the previous profiled optimum parallelism and thread mapping strategy persist. When a new optimum thread number is predicted and verified, a new CR range is prescribed. We choose 10%, as we consider 10% of the current performance change is good enough for the controller to choose a new strategy. We leave a better design of CR range decision function to the future work.

The four decision functions are illustrated in pseudocodes in Fig. 5 to further clarify their integration as well as their designs. The profile decision function is composed of two parts: disable and enable profile actions.

```
1  /*CR range decision func and parallelism predictor*/
2   thread_predition_func(){...}//see previous derivation
3   CR_range_func(){...}//see previous derivation
```

```
1  /*thread mapping and disable profile decision funcs*/
2  thread_mapping_prediction()
3  {
4    for i in {Linux,Scatter,Compact,RR} //mapping strategy
5    {
6      apply i;
7      if (current throughput> max throughput)
8         optimum mapping= i;
9    }
10   apply optimum mapping;
11 }
12 if (current throughput > max throughput)
13     compute a new CR range;
14     max throughput=current throughput;
15     if (optimum tn <= half the core number && tn changes )
16        apply thread mapping prediction;
17     else
18        disable profile;
19 else
20     optimum tn = previous tn;
21     apply previous optimum thread mapping;
22     disable profile ;
```

```
1  /*enable profile decision fuc*/
2   enable_profile_action();
3  {
4    record current mapping strategy;
5    record current max throughput;
6    record current tn;
7   }
8  if CR falls out the CR range
9     enable thread profile action;
10 else
11    keep collecting profile info;
```

Fig. 5. The main structure of the four decision functions. tn and thread mapping stand for the thread number and thread mapping strategy respectively.

It is worth noting that, in case the upper CR threshold is 100%, and the program CR is 100% (when only read operations in the transactions or no conflicts among transactions), higher parallelism to the program is assigned.

IV. Implementation

There are two methods for collecting application profile information in a parallel program. A master thread can be employed to record the interesting information of itself. An alternative way is to collect the information from all threads. The first method requires little synchronization cost for information gathering but the obtained information may not represent the global view. Also the master thread must be active during the whole program execution possibly resulting in earlier termination than the other threads, meaning that the fair execution time among threads can not be guaranteed. The latter method may suffer from synchronization cost but the profile information represents the global view. More importantly, a strategy for fair execution can be employed among threads. We choose the second method. The synchronization cost of information gathering is negligible for most of our applications.

We have implemented a monitor to collect the profile information, control the dynamic parallelism and the race condition. This implementation requires no modification to the applications. The monitor is a cross-thread lock which consists of the concurrent-access variables by threads. The major variables of the monitor are commits, aborts, two FIFO queues recording the suspended and active threads, the current active thread number, the optimum thread number and the throughput. There are three entry points of the monitor. The first entry point is upon threads initialization, where some initial values (*e.g.*, thread id) are set for the threads and all the threads pass the entry. The second entry point is upon a transaction committing, where commits are accumulated and where the control functions take actions. The third entry point is upon a thread exiting, where one suspended thread is awaken when one thread exits.

A time overhead is added to each transaction when calling and releasing a monitor. The overhead caused by calling lock is negligible on the transaction with medium and long length. But this overhead is significant for the transaction with a small number of operations. This is the case for **intruder** and **ssac2** (two applications of **STAMP** described in the later section). Such an overhead can be reduced through diminishing the frequency of calling the monitor, *i.e.* the monitor is called every 100 commits rather than every commit.

To guarantee fair execution time of threads, threads are periodically suspended and awaken. However this procedure brings thread migration which is costly. To reduce the influence of thread migration but meanwhile avoid thread starvation, the new awaken thread is mapped to the core where a thread is just suspended. Therefore only one thread is migrated each time rather than all the active thread remapping.

V. Performance Evaluation

In this section, we present the results from 6 different **STAMP** [12] benchmarks and one application from **EigenBench** [13]. The data sets cover a wide range from short to long transaction length, from short to long program execution time, from low to high program contention. The data sets also incorporate the applications with a unique phase and diverse online phases. Table I presents the qualitative summary of each application's runtime transactional characteristics: length of a transaction (number of instructions per transaction or

execution time per transaction), execution time (application execution time), and contention. The classification is based on the application with its static optimal parallelism. A transaction with execution time between 10*us* and 1000 *us* is classified as medium-length. The contention between 30% and 60% is classified as medium. The execution time between 10 seconds and 30 seconds are classified as medium.

TABLE I. Qualitative summary of each application's runtime transactional characteristics. Tx length is the number of instructions per transaction. Execution time means the whole program execution time. Contention is the global contention of the application. The classification is based on the application with its optimal parallelism applied.

Application	TX length	Execution time	Contention
EigenBench	medium	long	medium
intruder	short	medium	high
genome	medium	short	high
vacation	medium	medium	low
ssca2	short	short	low
yada	medium	medium	high
labyrinth	long	long	low

A. Platform

We evaluate the performance on a SMP machine with 4 processors of 6 cores each. Every pair of cores share a L2 cache (3072KB) and every 6 cores share a L3 cache (16MB). This machine holds 2.66GHz frequency and 63GB RAM. We utilise TinySTM as our STM platform.

B. Benchmark Settings

We have tested 6 different applications from **STAMP**, namely **intruder**, **ssca2**, **genome**, **vacation**, **yada** and **labyrinth**. Two applications, namely **bayes** and **kmeans** from **STAMP**, are not taken into account in this paper. As **bayes** exhibits non-determinism [14]: the ordering of commits among threads at the beginning of an execution can dramatically affect the execution time. The aforementioned applications have represented a wide range of characteristics of TM applications, therefore we do not present the results from **kmeans** due to the page limit. The inputs of the six selected applications are detailed in Table II.

TABLE II. The inputs of **STAMP**

intruder	-a8 -l176 -n109187
ssca2	-s20 -i1.0 -u1.0 -l3 -p3
genome	-s32 -g32768 -n8388608
vacation	-n4 -q60 -u90 -r1048576 -t4194304
yada	-a15 -i inputs/ttimeu1000000.2
labyrinth	-i random-x1024-y1024-z7-n512.txt

As most of the transactions within each **STAMP** application usually have very similar behaviour [4] which are not suitable for the evaluation of our dynamic thread mapping approach. To evaluate the efficiency of dynamical thread mapping strategy, we use **EigenBench** [13] (an artificial but highly configurable benchmark suit) to create new TM applications with diverse phases. When its parallelism varies, the corresponding thread mapping strategy varies. **EigenBench** includes 3 different arrays which provide the shared transactional accesses (Array1), private transactional accesses (Array2) and non-transactional accesses (Array3). We provide different read and write accesses to the three arrays to create the phase diversity. Due to the page limit, we only present the results on one application from **EigenBench**. The application gives two online phases. Its inputs are given in Table III.

TABLE III. The data set of **EigenBench** inputs for 24 threads.

		*R1	0	35
		*W1	0	45
loops	3333	*R2	0	200
A1	95536	*W2	0	100
A2	1048576	*R3o	0	10
A3	819200	*W3o	0	10
NOPi	0	*R1	1	300
NOPo	0	*W1	1	220
Ki	1	*R2	1	100
Ko	1	*W2	1	50
LCT	0	*R3i	1	0
M	2	*R3o	1	0
		*W3o	1	0

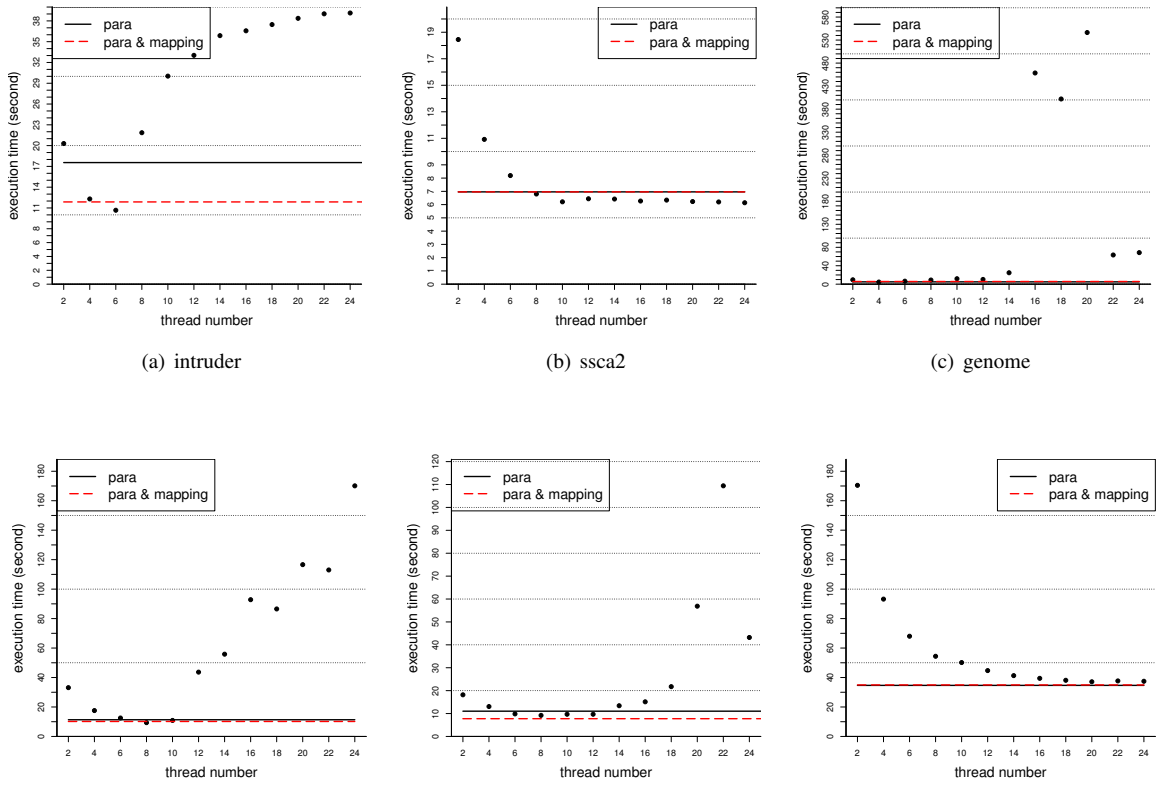
C. Results

We firstly present the results of the execution time comparison between two autonomic thread adaptation approaches with static parallelism. The maximum parallelism is 24 which is the number of the available cores. The minimum parallelism is 2, as we are only concerned with parallel applications. All the applications are executed 10 times with one application executing each turn. Simultaneous execution of multiple applications, which are more common in interactive systems, are not concerned. The results are the average execution time. In the following paragraphs, we address the model which only adjusts parallelism as *dynamic parallelism model*. And we address the model which regulates both parallelism and thread mapping strategy as *dynamic thread control model*.

Fig. 6 and Fig. 7 illustrate the execution time comparison with different static parallelism, *dynamic parallelism model* and *dynamic thread control model* of **EigenBench** and **STAMP**. The dots represent the execution time with different static parallelism. The solid black line stands for execution time with the *dynamic parallelism model* and the dashed red line gives the execution time with the *dynamic thread control model*. Fig. 7(b) indicates the performance comparison up to 10 threads to better illustrate the performance difference between two dynamic models on **EigenBench**,

According to Fig. 6 and Fig. 7, our two models outperform the performance of the majority of the static thread number. The *dynamic thread control model* shows positive performance rise against the *dynamic parallelism model* on applications: **EigenBench**, **yada** and **intruder**, but it indicates a performance degradation on **genome** and **vacation**. Both thread models bring the similar performance to **labyrinth** and **ssca2**. Table IV and Table V detail the performance comparison. The digits in the brackets are the static parallelism which gives the best and the worst performance respectively. Both thread models outperform the best case of static parallelism on **EigenBench**, **genome** and **labyrinth**. Both thread models shows performance degradation on **vacation**, **intruder** and **ssca2** against the best case, yet brings significant performance improvement comparing with the average value and the worst case. It is worth noting that, some of TM applications scale poorly when their parallelism rises (*e.g.*, **genome**, **vacation**, **EigenBench**) due to TM mechanisms.

Fig. 8 demonstrates the parallelism variation for both models and the thread mapping strategy variation for the *dynamic*



(a) intruder (b) ssca2 (c) genome

(d) vacation (e) yada (f) labyrinth

Fig. 6. Time comparison for **STAMP** for static parallelism, *dynamic parallelism model* and *dynamic thread control model* . The dots represent the execution time with different static parallelism.

TABLE IV. Performance comparison on different applications with *dynamic parallelism model*. The performance is compared with the minimum, average and the maximum value of all the static parallelism.

benchmarks	best case	average value	worse case
EigenBench	+14% (2)	+95%	+99% (24)
genome	+16% (4)	+97%	+99% (20)
vacation	-18% (8)	+83%	+94% (24)
labyrinth	+8% (24)	+43%	+80% (2)
yada	-21% (8)	+59%	+90% (22)
ssca2	-13% (24)	+12%	+62% (2)
intruder	-64% (6)	+41%	+55% (24)

TABLE V. Performance comparison on different applications with *dynamic thread control model*. The performance is compared with the minimum, average and the maximum value of all the static parallelism

benchmarks	best case	average value	worse case
EigenBench	+25% (2)	+96%	+99% (24)
genome	+1% (4)	+97%	+99% (20)
vacation	-33% (8)	+80%	+93% (24)
labyrinth	+7% (24)	+42%	+80% (2)
yada	+16% (8)	+72%	+93% (22)
ssca2	-14% (24)	+11%	+62% (2)
intruder	-11% (6)	+60%	+70% (24)

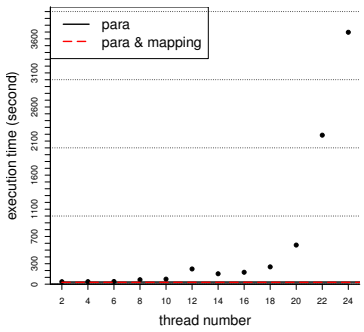
thread control model. As shown in the figures, the **EigenBench** application shows thread mapping strategy variation, whereas **yada** keeps the same strategy during its whole execution time.

Lastly to validate the correctness of our approaches on prediction of suitable parallelism and thread mapping strategy at each phase, we present the online throughput variation comparison as shown in Fig. 9. Due to the page limit, we only present the results on two applications. Both dynamic models rival or exceed the maximum throughput of the static parallelism on different phases.

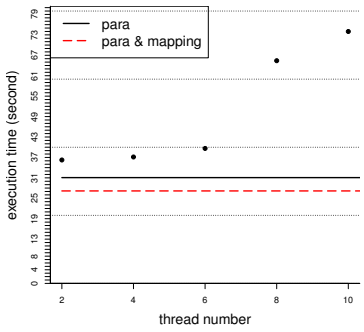
VI. Discussion

The overhead of our approaches mainly originate in three aspects: (1) Thread migration. This stems in two points: switching among different thread mapping strategies and periodically awake and suspend the threads to ensure the fair execution time for each thread. (2) Unnecessary profiling of thread mapping strategies. Most of the transactions within **STAMP** applications show very similar behaviour, therefore the thread mapping strategy is only needed to be profiled once despite of the altered parallelism later. The more frequently the parallelism is adjusted, the higher overhead the applications suffer. (3) The cost of calling locks. Two factors contribute to this cost: the operation of obtaining and releasing the lock and the time spent contending the lock. The latter cost increases significantly with higher active number of threads and gives significant impacts on the applications with short-length transactions. However this cost is trivial as explained in Section IV.

We utilise a simple CR range decision function to determine application phase changes. The decision function shows its limitation on phase detection for **intruder**. As its CR tends

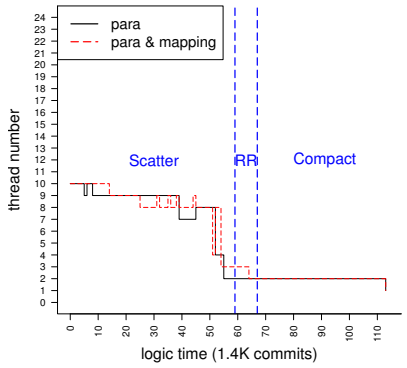


(a) all threads

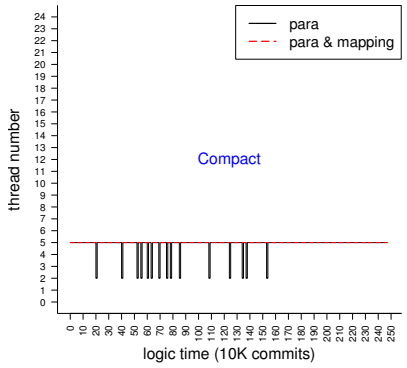


(b) up to 10 threads

Fig. 7. Time comparison of **EigenBench** for static parallelism, *dynamic parallelism model* and *dynamic thread control model*. The dots represent the execution time with different static parallelism.



(a) **EigenBench**



(b) **yada**

Fig. 8. Parallelism and thread mapping strategy variation at runtime. The solid black line is by *dynamic parallelism model*, the dashed red line is by *dynamic thread control model*.

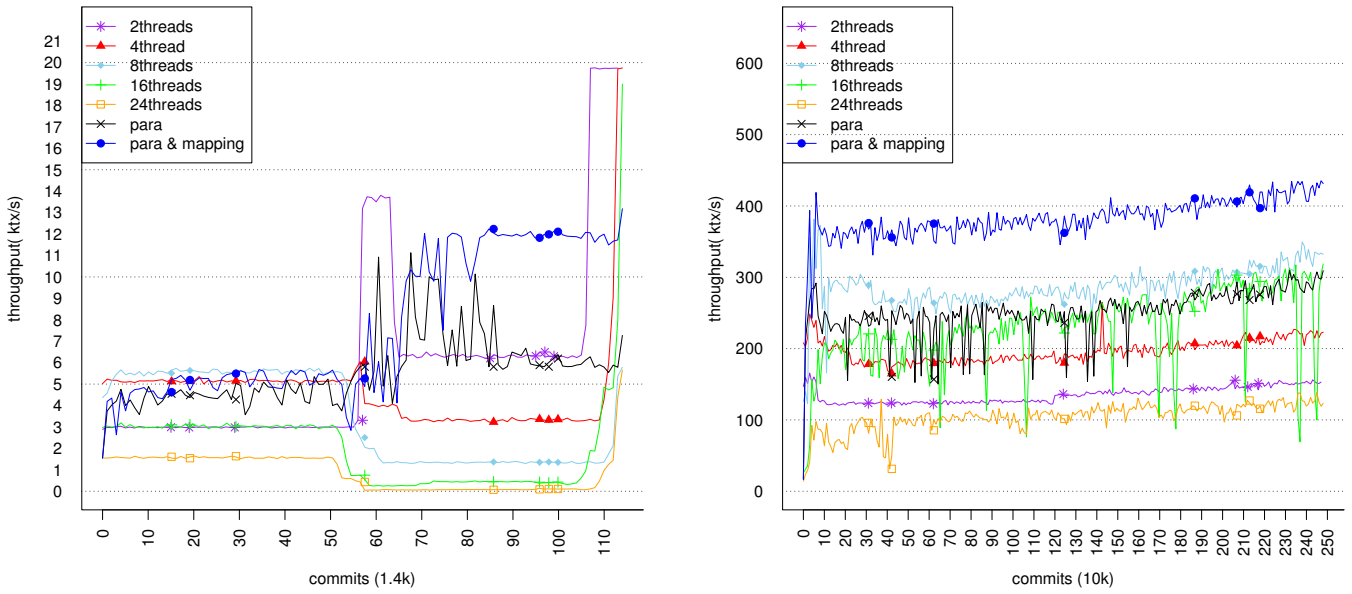
to fluctuate frequently over the CR range, yet remains in the same phase. Therefore both dynamic models overreact to such changes especially the *dynamic parallelism model*, but *dynamic thread control model* better controls thread migration and prevents abrupt parallelism change thus performs better than the other model.

The parallelism predictor relies on two assumptions which are based on ideal situations, thus some errors are imposed inevitably in reality, meaning that the predicted parallelism may be slightly different as the optimum one (*e.g.*, **yada**). However such performance lost is compensated by the performance gain from thread mapping. The gain obtained with a dynamic approach over a static one is directly related to the diversity of application phases. The more distinct are the phases, the higher gain is. Hence our proposed dynamic approaches are more interesting to the applications with online behaviour variations. Although the dynamic framework is capable of optimizing the parallelism and thread mapping strategy for the applications with stable online behaviour, the performance benefit can not always compensate the profile overhead, *e.g.*, **ssca2**. It is also worth noting that not all the applications require to set a thread mapping strategy. For instance the application with low contention and its parallelism degree equals to the core number (*e.g.*, **labyrinth**, **ssca2**). Therefore, both models illustrate similar performance on the two applications. Additionally, profiling the thread mapping strategy gives penalty to **genome** and **vacation** as the two applications contain sudden contention changes which the *dynamic parallelism model* can respond immediately. But the *dynamic thread control model* requires extra profiling lengths to search a better thread mapping strategy, meanwhile the applications have already entered a new phase.

Compact is favoured when CR is low and **Scatter** is likely to be selected when CR is high. But when CR is high, the *dynamic thread control model* favours higher parallelism. When the parallelism is approaching the maximum core number, the thread mapping strategy profiling is disabled as little performance impact can be received from thread mapping.

VII. Conclusion and Future Work

In this paper, we investigate autonomic parallelism adaptation and thread mapping regulation on a STM system. We examine the performance of different static parallelism and concluded that runtime regulation of parallelism and thread mapping is necessary to the performance of STM systems. We then present our approaches and compared their performance with static parallelism. We introduce a feedback control loop to manipulate the threads at runtime. We then analyse the implementation overhead and discuss the advantages as well as limitations of our work.



(a) **EigenBench** (b) **yada**

Fig. 9. Online throughput variation. The *dynamic parallelism model* and *dynamic thread control model* are the black line with crosses and the blue line with dots respectively.

Thread migration among cores impacts on system performance and causes performance degradation. Four different thread mapping strategies are profiled in order to select the optimum one. Such a profiling procedure is costly, as it brings thread migration and also imposes the program to work partly under an unsuitable thread mapping strategy. We plan to design a thread mapping strategy predictor which can predict the optimum thread mapping strategy in one step. We present a general approach on thread control for STM system which can be transferred to HTM systems to obtain better performance by taking advantage of hardware support in future work.

ACKNOWLEDGMENT

This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir.

REFERENCES

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," *SIGARCH Comput. Archit. News*, vol. 21, pp. 289–300, May 1993.

[2] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, (New York, NY, USA), pp. 237–246, ACM, 2008.

[3] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," *SIGPLAN Not.*, vol. 41, pp. 336–346, Oct. 2006.

[4] M. B. Castro, *Improving the Performance of Transactional Memory Applications on Multicores: A Machine Learning-based Approach*. PhD thesis, University de Grenoble, December 2012.

[5] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, pp. 41–50, Jan. 2003.

[6] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing — degrees, models, and applications," *ACM Comput. Surv.*, vol. 40, pp. 7:1–7:28, Aug. 2008.

[7] 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*, Euro-Par '08, (Berlin, Heidelberg), pp. 719–728, Springer-Verlag, 2008.

[8] D. Rughetti, P. Di Sanzo, B. Ciciani, and F. Quaglia, "Machine learning-based self-adjusting concurrency in software transactional memory systems," in *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pp. 278–285, Aug 2012.

[9] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker, "Identifying the optimal level of parallelism in transactional memory applications," in *Networked Systems* (V. Gramoli and R. Guerraoui, eds.), vol. 7853 of *Lecture Notes in Computer Science*, pp. 233–247, Springer Berlin Heidelberg, 2013.

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

[11] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: A machine learning based approach," *SIGPLAN Not.*, vol. 44, pp. 75–84, Feb. 2009.

[12] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *2008 IEEE International Symposium on Workload Characterization (IISWC)*, September 2008.

[13] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "EigenBench: A simple exploration tool for orthogonal TM characteristics," in *2010 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–11, Dec 2010.

[14] W. Ruan, Y. Liu, and M. Spear, "STAMP need not be considered harmful," in *9th ACM SIGPLAN Workshop on Transactional Computing*, (Salt Lake City), March 2014.