# An Autonomic-Computing Approach on Mapping Threads to Multi-cores for Software Transactional Memory

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

# An Autonomic-Computing Approach on Mapping Threads to Multi-cores for Software Transactional Memory

Naweiluo Zhou[1,2,3,5], Gwenaël Delaval[1,2,3,5], Bogdan Robu[1,4,5]
Éric Rutten[3], Jean-François Méhaut[1,2,3,5]

January 28, 2025

Univ. Grenoble Alpes[1], LIG[2], INRIA[3], GiPSA-Lab[4], CNRS[5], F-38000, Grenoble, France
Email:(naweiluo.zhou, jean-francois.mehaut)@imag.fr,
(gwenael.delaval, eric.rutten)@inria.fr, bogdan.robu@gipsa-lab.fr

**Abstract**

A parallel program needs to manage the trade-off between the time spent in synchronisation and computation. This trade-off is significantly affected by its parallelism degree. A high parallelism degree may decrease computing time while increasing synchronisation cost. Furthermore, thread placement on processor cores may impact program performance, as the data access time can vary from one core to another due to intricacies of the underlying memory architecture. Alas, there is no universal rule to decide thread parallelism and its mapping to cores from an offline view, especially for a program with online behaviour variation. Moreover, offline tuning is less precise. We present our work on dynamic control of thread parallelism and mapping. We address concurrency issues via Software Transactional Memory (STM). STM bypasses locks to tackle synchronisation 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 implement feedback control loops to automate management of threads and diminish program execution time.

**Keywords:** autonomic computing, transactional memory, feedback control, synchronisation, parallelism, thread mapping

# 1 Introduction

Multi-core processors accelerate computation using high thread parallelism (number of simultaneously active threads). A program for multi-cores executes in parallel and needs to scale when the number of cores increases. However, writing a parallel application is difficult, as parallel programming encompasses all the difficulties of sequential programming and introduces extra problems on coordination of interactions among concurrently executing tasks [1]. High thread parallelism may shorten execution time, but it may potentially increase synchronisation time.

Moreover, multi-core processors incorporate complex memory hierarchies, which consist of several levels of cache to alleviate penalties caused by the data access to main memory. Consequently, parallel applications need to evolve to efficiently exploit the potential of their underlying architecture. Depending on the level of cache where data are placed, access latency differs for different cores. To alleviate access latency, threads can be fixed to certain cores to improve their resource usage, *e.g.* cache, main memory and interconnections.

The conventional way to address synchronisation is via locks. However, locks are notorious for various issues such as deadlocks as well as the vulnerability to thread failures [2, 3]. Moreover, it is not straightforward to analyse interactions among concurrent operations. Transactional memory (TM) has emerged as an alternative parallel programming technique that handle synchronisation through transactions rather than locks [4]. Access to shared data is enclosed in transactions that are speculatively executed without blocking by locks. Various TM schemes have been developed including Hardware Transactional Memory (HTM) [4], Software Transactional Memory (STM) [5] and Hybrid Transactional Memory (HyTM) [6]. This paper presents the work on thread management under STM systems where the synchronisation time mainly originates from transaction aborts. There are different ways to reduce the number of aborts [2], such as the design of contention management policies (resolve conflicts among transactions), the way to detect conflicts (detect at early stage or later stage), the setting of version management (handles the storage policy for permanent and transient data copies) and the level of thread parallelism.

Online parallelism adaptation has recently begun to receive attention for STM. It is onerous to determine a parallelism degree offline especially for the one with online behaviour fluctuations [7, 8]. Therefore, the natural so-

lution consists of monitoring the program online and altering its parallelism when necessary. Furthermore, application performance is affected by diverse placements of threads. When the active thread number varies, locations that threads are pinned to may also need to be adjusted accordingly in order to optimise usage of the memory hierarchy. Pinning multiple threads to specific cores is called *thread mapping* [9]. One previous work [9] has presented approaches on adjusting thread mapping online for STM. Alas, previous studies for TM either only addressed adaptation of online parallelism or thread mapping.

The diversity of TM applications and their supporting TM platforms together with the complexity of multi-core processor architecture make it difficult to decide the configurations of various parameters offline. Dynamic interactions among applications, TM platforms and underlying hardware can impact system performance. All the aforementioned issues are preferred to be dealt with online. Autonomic computing [10] is a technique that can automatically manage systems given high-level objectives. We introduce feedback control loops into STM systems to achieve autonomic computing, more specifically, to automatically regulate thread parallelism and mapping online. Literature [11, 7, 12, 13, 14, 15, 16, 17, 18, 19] has shown insight into implementing feedback control loops to regulate online parallelism degree for TM. Their methods either depend on offline training data or takes long profiling time online. In this paper, we present effective frameworks for thread management on TinySTM [5]. The main contributions of our paper are as follows:

1. We illustrate two approaches that automatically adjust a program to its near-optimal parallelism degree in order to improve system performance online.

2. We propose two phase detection functions.

3. We present an approach which can adjust both thread parallelism and mapping through coordination of feedback control loops.

The rest of this paper is organized as follows. Section 2 presents the relevant background. Section 3 details the profiling procedures and the online parallelism adaptation methods. Next, Section 4 gives the approach which integrates the control of thread parallelism and mapping. Section 5 shows the implementation details. The results are illustrated in Section 6. Section 7 discusses the limitations of our frameworks. Section 8 reviews the related work. Lastly, Section 9 concludes the paper and proposes future work.

# 2 Background

## 2.1 Background on Software Transational Memory

Transactional memory (TM) is an alternative synchronisation technique. In TM, data access to shared memory is 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 [4]. Two metrics are often used in TM to indicate system performance, namely *commit ratio* and *throughput*. Commit ratio (CR) [11, 7, 12] 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. 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 with a combination of the two (hybrid). Different mechanisms explore the design trade-off that impacts performance, programmability and flexibility. In this paper, we focus on STM systems and utilise TinySTM [5] 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. Their locks are utilised to indicate ownership of transactions rather than stalling threads.

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 [20], 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. Higher contention in a program leads to larger amount of wasted work. The time spent in wasted work is the synchronisation time in the STM view.

## 2.2 Thread Parallelism and Mapping

Apart from diminishing wasted work, one way to improve STM system performance is to trim computing concurrency. High parallelism may accelerate computation but resulting in high contention, thus high synchronisation cost. Hence parallelism degree can significantly affect performance of a program. Furthermore, 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 the application's performance. Fig. 1 illustrates the access latency from the core to different memory levels[1].

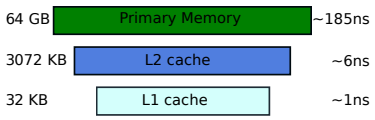



Figure 1: An illustration for access latency of diverse memory levels.

Depending on the number of cores and the memory hierarchy, there can be many different strategies to implement thread mapping. Using exhaustive search (which is utilised for optimisation on some conventional parallel program) by trying all the cases to analyse the best placement strategy is not feasible, especially when a large number of cores are involved. On a unified memory system, we can categorise thread mapping strategies into four main groups [22] based on cache sharing.

- **Compact**: threads are placed on sibling cores. This strategy can benefit the applications whose threads possess a significant amount of joint data access. Placing threads on sibling cores which share all the levels of memory structure allows threads to reuse the data which already resides in the cache. Threads sharing data being scheduled on the cores which do not share cache can result in excessive data movement and high network traffic [23].

- **Scatter**: threads are distributed across processors. Equal distribution of threads is also addressed as *thread balancing* [24]. This strategy averts cache sharing among cores in order to reduce contention on the same cache. Applications that show disjoint data access may benefit from this strategy.

- **Round-Robin**: threads are placed on the cores where a higher level of cache (*e.g.* L3) is shared but not the lower level of cache (*e.g.* L2).

[1]The latency is measured by **lmbench** [21] on our platform. The L3 cache latency is skipped in the figure.

This strategy can only be applied on the platforms where the L2 cache is shared by more than one core.

- **Linux**: the default Linux scheduling strategy. It is based on dynamic priority-based strategy that allows threads to migrate to idle cores to balance the run queues.

## 2.3 Background on Autonomic Computing

Autonomic computing [25], proposed by IBM in 2003, is a concept that brings together many fields of computing with the purpose of creating self-managed computing systems. Autonomic computing proposes a general structure of feedback loop to take adaptive and reconfigurable computing into account [26]. Feedback control loops, on one form or another, have been adopted as cornerstones of software-intensive and self-adaptive systems [27, 28, 29]. A classic feedback control loop is illustrated in Fig. 2 in the shape of a MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) loop.

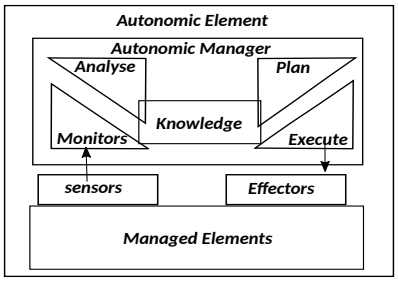
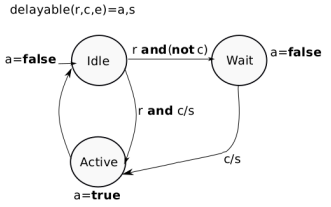



Figure 2: A MAPE-K control loop. It incorporates an autonomic manager, sensors, effectors as well as managed elements among which the autonomic manager plays the main role.

In general, a feedback control loop is composed of (1) an autonomic manager (also known as a controller in control theory and control engineering), (2) sensors (to collect information), (3) effectors (to carry out changes), (4) managed elements (any software or hardware resource). An autonomic manager is composed of five elements: a monitor (used for sampling input data), an analyser (to analyse data obtained from the monitor), knowledge (knowledge of the managed system), plan (to utilise the knowledge of the system to carry out computation) and execute (to perform changes). The above five elements of the autonomic manager can overlap with each other.

### 2.3.1 Synchronous Programming language–Heptagon

We utilise Heptagon to implement the controller. Heptagon [30, 31] is a synchronous programming language which allows us to describe reactive systems by means of generalised Moore machines, *i.e.* mixed synchronous dataflow equations and automata with parallel and hierarchical composition. Fig. 3 illustrates a small program which is referenced as a node in Heptagon. This program defines a task (*delayable*) that can be either idle or active. The program calls one state at each reactive step. It remains in the *Idle* state until the occurrence of the input $r$ requests the launch of the task. Another input $c$ (which will be controlled by another controller) can either allow the activation, or temporarily block the launch request, thus leading the automaton to shift to a waiting state (the *Wait* state). When active, the task can terminate and return to the *Idle* state upon the notification input $e$. This task yields two outputs: a representing activity of the task $a$, and $s$ being emitted at the instant when it becomes active. The two outputs are the control decisions, the effectors receive the two outputs and make control actions to the managed elements.




```
1   node delayable(r,c,e:bool) returns(a,s:bool)
2    let
3     automaton
4       state Idle
5       do a = false ; s = r and c
6       until r and c then Active
7         | r and not c then Wait
8       state Wait
9       do a = false ; s = c
10      until c then Active
11      state Active
12      do a = true ; s=false
13      until e then Idle
14    end
15  tel
```

Figure 3: A *delayable* task in graphical and textual syntax.

# 3 Autonomic Parallelism Adaptation

In this section, we detail the design of two feedback control loops to dynamically determine near-optimum parallelism. We firstly introduce a *simple model* which searches near-optimum parallelism, then we present a more sophisticated model (*probabilistic model*) based on the probability theory which predicts the near-optimum parallelism.

We measure three metrics from the STM system, namely the number of commits, the number of aborts and execution time. The number of commits and the number of aborts are addressed as *commits* and *aborts* subsequently.

We choose CR (its value is between 0 and 1) and throughput to denote program performance, as CR and throughput are both sensitive to parallelism variation. Either is, by itself, not sufficient to represent program performance as:

- A high throughput shows fast program execution whereas a low throughput indicates slow program progress. Nevertheless, a low throughput may be caused by low parallelism or by a low number of transactions taking place.
- CR indicates the conflicts among threads. A high CR means low synchronisation time whereas a low CR means high synchronisation cost for most STM. However, 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 are running.

We continuously measure the CR to detect contention fluctuation and enable corresponding control actions. The correctness of the control actions are verified by checking if the throughput is improved after the actions for parallelism adaptation.

## 3.1 Overview of the Profiling Algorithm

We firstly give an overview of the profiling procedure and later describe the *simple model* and *probabilistic model* through the prism of control theory. The profiling algorithm is also shared by the approach presented later in Section 4.

To achieve autonomic parallelism adaptation which provides a program with its optimum parallelism, we propose to periodically explore the parallelism and select the value that achieves the highest throughput. By observing CR, we can obtain the contention information of programs. CR usually fluctuates in a certain range within the same phase. When the current parallelism produces a different CR which falls out of the current CR range, the program enters a new phase. The CR fluctuation can trigger a new parallelism adaptation action. In an application, CR always diminishes with increment of parallelism degree due to increase of conflicts among threads. A notable exception would be an application with few write transactions making its CR always remain 1. Initially, the upper and lower CR thresholds of the CR range are both set to be 0 and are trained in the later profile stage. The detail of the profiling procedure is illustrated in Fig. 4.

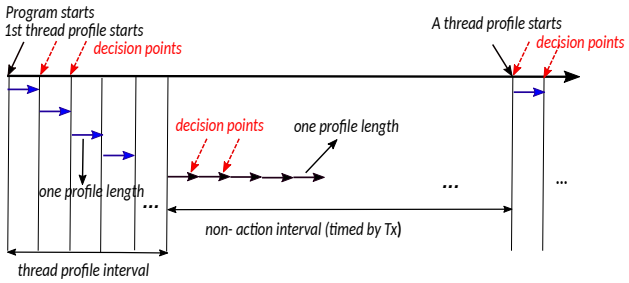
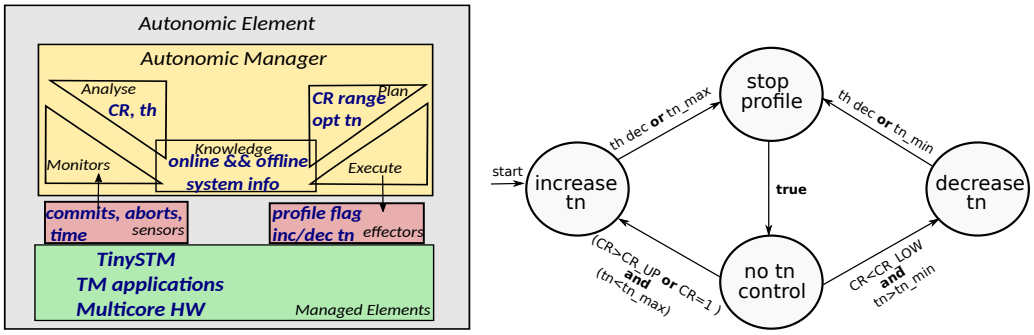



Figure 4: Periodic profiling procedure. At each decision point, the control actions are taken.

The profiling procedure starts once the program starts. Initially the program creates a pool of threads, among which only 2 threads are awakened and the rest are suspended in the *simple model*. *Per contra*, all threads are set to be active in the beginning in the *probabilistic model*. At each decision point, which corresponds to one state of the automaton in Fig. 5(b), the control loops (see Section 3.2 and Section 3.3) are activated to adapt the parallelism or suspend the parallelism adaptation. A *profile length* is a fixed number of commits. At the end of one profile length is the *decision point*. A *thread profile interval* is composed of a continuous sequence of profile lengths, within which the parallelism is adjusted and the program phase is computed. The *non-action interval* consists of one or a continuous sequence of profile lengths, within which the parallelism regulation is suspended. The duration of thread profile interval and non-action interval are not fixed values as indicated in Fig. 4. The choice of the profile length mainly depends on the total amount of transactions in an application. The applications with the same magnitude of transactions share the same profile length. This is further explained in Section 5.

## 3.2 Feedback Control Loop of the Simple Model

Fig. 5(a) gives the structure of the complete platform that is an instantiation of a MAPE-K feedback control loop. The autonomic element is composed of the STM system, benchmarks, inputs, outputs and the autonomic manager. The autonomic manager, which can be also regarded as the controller, is described as an automaton as shown in Fig. 5(b). This automaton consists of four states, and one state is called at each decision point corresponding to Fig. 4.




(a) The instantiation of MAPE-K-shape feedback control loop for *simple model*.

(b) The structure for the autonomic manager of Fig. 5(a) in automaton shape.

Figure 5: The feedback control loop of the *simple model*. th stands for throughput and tn means the number of thread. The boolean value **true** means unconditional state shift.

### 3.2.1 Control Objective

Under control theory terminology, the control objectives of our feedback control loop are to maximise throughput and diminish global execution time.

### 3.2.2 Inputs and Outputs

As shown in Fig. 5(a), the inputs are commits, aborts and execution time. The outputs, which are also known as control actions, are increment or decrement of parallelism degrees and settings on profile flags which enable or disable control actions.

### 3.2.3 Decision Functions

Three decision functions cooperate to take decisions: a *parallelism decision function* (adjusts parallelism), a *profile decision function* (enables the actions of adjusting parallelism) and a *phase decision function* (computes a CR range which determines the program phase). The first two functions are given in the next paragraph and the phase function is presented lastly in this section. We describe our parallelism controller as an automaton, since an automaton can elucidate the relations among the decision functions and how the decision functions are designed.

The automaton commences at the state *increase tn*, as the parallelism is set to be the minimum at the starting point. At each instance of *increase tn*, one thread is awakened if the current throughput is greater than the maximum throughput that is recorded in the current thread profile interval. The

initial value of maximum throughput is 0. In each thread profile interval, the parallelism can either continuously increase or decrease. The action on whether to increment or decrement parallelism is determined by the *profile decision function*. The state shifts to *stop profile* when the throughput of the current instance is less than the maximum value. At the final decision point of a thread profile interval (at state *stop profile*), the parallelism is set to be the value which yields the maximum throughput. The automaton then shifts from *stop profile* to *no tn control* which corresponds to the non-action interval in Fig. 4. At each decision point of a non-action interval, the profile decision function checks if the controller needs start parallelism adaptation. More specifically, when the CR falls into a certain range, the program remains in *no tn control* state. Otherwise a boolean value is set indicating the direction of the parallelism regulation (increment or decrement). The automaton shifts its state to *increase tn* if CR is higher than the upper CR threshold. Otherwise the state shifts to *decrease tn* when CR is less than the lower CR threshold. At each instance of *decrease tn*, one thread is suspended. The automaton remains in *decrease tn* when the throughput shows improvement at each instance. In case the value of the upper threshold is 1 and is identical to the current program CR, a higher parallelism degree is assigned to the program. This situation happens when only read operations or no conflicts across transactions.

The throughput often fluctuates before reaching the optimum value as shown in Fig. 6. To prevent a parallelism profiling procedure from terminating at a local maximum throughput, parallelism profiling procedure continues until the throughput decreases over 10% of the maximum value recorded. 10% is an empirical value based on results analysis which can be tuned. This value is consistent with the value of $\delta$ in Section 4.2.

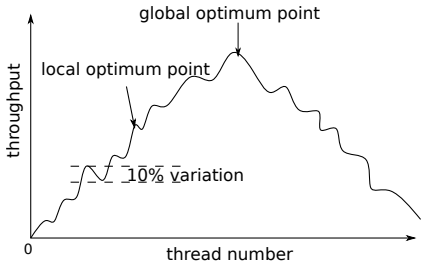



Figure 6: Throughput fluctuation. The throughput may continuously rise and descend before reaching its maximum point.

The *parallelism decision function* is called at state *increase tn* and *decrease tn*. Parallelism adaptation is activated when a program enters a new phase. A new phase is denoted by when CR fluctuates out of a certain

range. It is onerous to determine such a CR range offline, especially for some programs with online performance variation. Additionally, a constant CR range impedes programs to search its optimum parallelism. Therefore, it becomes necessary to dynamically resolve a CR range. We add a *phase decision function* to reach this goal which is called at *stop profile* state. The phase decision function is called at the end of a thread profile interval where a new CR range is prescribed. The two thresholds of the CR range are the CR values produced by running with one more or one less parallelism degree than the optimum. We refer to this CR range decision function as *simple phase decision function*.

## 3.3 Probabilistic Model

The approach to manipulate the parallelism degree by one at each decision point can engage long profiling time before the optimum value is decided. This section presents a *probabilistic model* which predicts a favourable parallelism degree after one profile length based on the CR and current active thread number.

The *probabilistic model* shares the same inputs, outputs and control objectives as the *simple model*. It also incorporates three decision functions. Since the *phase decision function* is equivalent to that in Section 3.2.3, we only describe the *parallelism decision function* and the *profile decision function* in this section.

### 3.3.1 Decision Functions

We firstly present the *parallelism decision function*.

Let $L_0$ be a fixed period. Assuming that the average length of transactions (including the aborted and committed transactions) is $L$ and the current number of active thread is $n$, then the number of transactions $N$ ($N = commits + aborts$) executed during the $L_0$ period can be expressed as: $N = \frac{L_0}{L} \cdot n = \alpha \cdot n$.

A transaction can commit during the $L_0$ period if it encounters no conflict with the rest of concurrently active transactions. We assume that the conflict probability $p$ between two transactions is independent from the active threads at the current phase, thus independent from the number of active transactions. The transactions executed in a sequence within the same thread cause no conflicts among each other.

We assume that during the $L_0$ period, the executed transactions are homogeneous, and approximately of same length $L$. Then, during $L_0$, each

thread approximately executes the same number of transactions $\frac{N}{n}$. Therefore, during this same period, the number of transactions causing potential conflicts with one transaction is the total number of transactions $N$, minus the transactions on the current thread of this transaction: $N - \frac{N}{n}$.

For a transaction $i$, let $X_i$ be a random variable, with $X_i = 1$ if the transaction is committed, and $X_i = 0$ if it is aborted. By approximation of the execution of a STM program, we assume that a transaction is aborted if it conflicts with at least one other transaction during the $L_0$ period. Then, the probability of a commit can be expressed in Equation 1.

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

where $q = 1 - p$ stands for the probability of no conflict between two transactions.

Equation 1 can hold provided that there is a large amount of transactions executed during the $L_0$ period making the conflict probability $p$ between two transactions approach a constant.

Under the terminology of probability theory, $X_i$ is a random variable. $X_i$ follows a Bernoulli distribution with its parameter $q^{\alpha(n-1)}$. Let $T$ represent the throughput. In a unit of time, the throughput is equivalent to the commits, which can be expressed as $T = \sum X_i$. Further, CR can be expressed as $CR = \frac{T}{N}$. Hence $T$ follows a binomial distribution, $T \sim B(N, q^{\alpha(n-1)})$. The expected value of $T$ is:

$$E[T] = N \cdot q^{\alpha(n-1)} = \alpha n q^{\alpha(n-1)} \tag{2}$$

Hence the expected value of CR is:

$$E[CR] = \frac{E[T]}{N} = q^{\alpha(n-1)} \tag{3}$$

Equation 2 can be rewritten as a function from $n$ to $T$ as shown in Equation 4.

$$T(n) = \alpha \cdot n \cdot q^{\alpha(n-1)} \tag{4}$$

To obtain the value of $n$ where $T$ reaches the maximum, we compute the derivative of Equation 4: $T'(n_{opt}) = 0 \rightarrow n_{opt} = -\frac{1}{\alpha \ln(q)}$ , where $n_{opt}$ stands for the optimum parallelism degree.

From Equation 3, we can obtain $q = CR^{\frac{1}{\alpha(n-1)}}$. Then we can derive the following equation:

$$n_{opt} = -\frac{n-1}{\ln(CR)} \tag{5}$$

where $n$ represents the number of active threads. In case $CR = 0$, $n_{opt} = 0$. The further details of the derivation are stated in [32, 33].

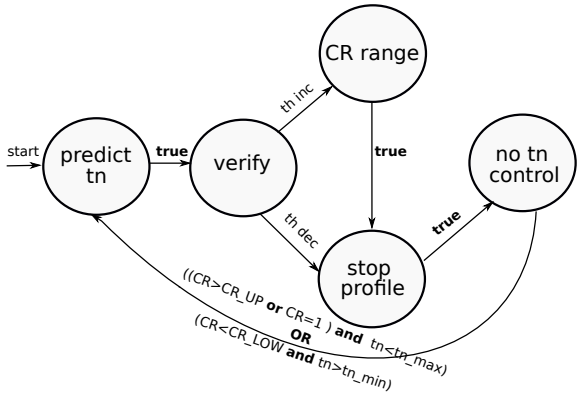



Figure 7: The controller structure of the *probabilistic model* described as an automaton. *th* and *tn* represent throughput and thread number, respectively.

In this paragraph, we describe an automaton to elucidate the relations of the three decision functions and the design of the profile decision function. As illustrated in Fig. 7, the automaton commences (with the maximum parallelism) from *predict tn* state which yields an estimation of the optimum parallelism degree. The predicted parallelism is applied for one subsequent profile length and the automaton unconditionally shifts its state to *verify* state. The new parallelism degree is only applied subsequently when the throughput during the verification time is larger than the throughput recorded before prediction. This leads the state to shift to *CR range* state where a new CR range is prescribed. The *stop profile* state disables the parallelism profile action either when the parallelism does not alter after verification or when a new CR range is resolved. Otherwise it recovers the previous value of the parallelism degree (CR range remains unchanged in this case). Contrary to the *simple model*, the *probabilistic model* requires an individual state to obtain the CR range. The *simple model* continuously increases or decreases the parallelism degree until the optimum throughput is reached, therefore, it requires no additional state for CR range decision.

# 4 Autonomic Thread Management

Apart from thread parallelism, thread mapping may impact program performance, as the data access time can vary from one core to another. Fixing threads on specific cores can diminish data access time, thereby reduce the

overall execution time of a program. This section will investigate the issues on thread mapping and its coordination with parallelism. We describe two feedback control loops which collaborate to improve system performance.

## 4.1 The Overview of MAPE-K Loop for Parallelism and Mapping Management

In this section, the instantiation of MAPE-K loop as illustrated in Fig. 5(a) is complemented with the control of thread mapping strategies. Accordingly, its effectors are capable of carrying out changes on thread mapping strategies. However, its overall structure is similar as in Fig. 5(a). We assign the same control objectives to the MAPE-K loop. It has the same inputs, hence the same sensors. The following three sections concentrate on the designs of the autonomic manager. Specifically, the autonomic manager incorporates two automata with one slave loop determining thread mapping strategy and one master loop deciding the parallelism degree as well as controlling the former loop.

The autonomic manager consists of two automata which include four decision functions cooperating to make decisions: *a parallelism decision function*, a *thread mapping decision function* (decides the thread mapping strategy), a *phase decision function* (computes the CR range) and a *thread profile decision function* (enables/disable the thread profile action). At each decision point as illustrated in Fig. 4, one corresponding decision function reacts to make its decision. We firstly detail the master loop which determines the parallelism degree and the program phase. Then we describe the coordination policy which elucidates the relations between the master and slave loops. The slave loop, which is the thread mapping decision function, is presented lastly.

## 4.2 The Master Loop

The master loop makes use of the *probabilistic model* as our parallelism decision function rather than the *simple model*, for the following reasons:

- The time spent in parallelism prediction is shorter. The *simple model* responds slowly to program phase changes and requires longer time for optimum parallelism detection. *Per contra*, the *probabilistic model* responds rapidly to phase variation and needs short thread profile time despite the fact that it may over-react to phase fluctuations.

- It shortens the thread profile interval. It requires extra profile lengths to decide the best mapping strategy. Together with the long profile

time imposed by the *simple model*, the thread profile interval is *de facto* at risk to exceed the length of a program phase.

Two control decisions need to be taken: thread parallelism degree and thread mapping strategy. This brings up the question on which decision should be made first. The parallelism degree can affect the choice of the best thread mapping strategy. Intuitively, the thread mapping strategy may in turn affect the parallelism prediction. Fig. 8 shows how the best mapping strategy varies with 2, 4 and 8 thread number for one application (*i.e.* **EigenBench**). **Linux** is the best choice when the parallelism degree are 2 and 4 and **Round-Robin** shows the best performance for 8 threads. Our MAPE-K loop chooses to predict parallelism prior to mapping under the scrutiny as stated below:

1. The prediction of the thread mapping strategy requires knowledge of the parallelism degree. Some TM applications do not scale with an increment of its parallelism degree regardless of the mapping strategy that is applied. It is unnecessary to predict the mapping strategy when the behaviour of a program is unstable.

2. The parallelism degree demonstrates more significant performance impacts than that of the thread mapping strategy, as later illustrated in performance evaluation in Section 6.

3. The impact of the thread mapping strategy on parallelism prediction is trivial. This is based on observation of application performance.

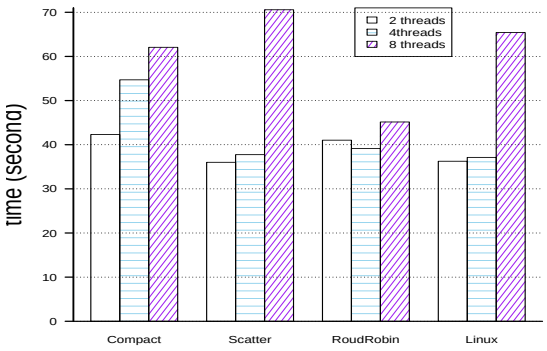



Figure 8: Performance difference for four mapping strategies when the parallelism degree varies.

The master loop, which is shown in Fig. 9, forms an automaton similar to the one in Fig.7, with the *CR range* state being substituted by the *mapping* state to dictate the optimum mapping strategy. The phase decision function is called at the state *stop profile*. Additionally, when the decision function yields identical parallelism degree as the previous result, the automaton shifts to the state *no profile* directly without shifting to the state *verify*.

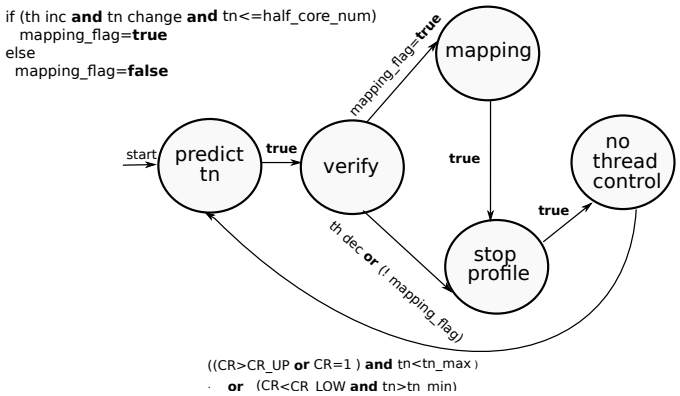



Figure 9: The automaton for parallelism and mapping decision. *th* and *tn* represent throughput and thread number, respectively. The boolean value **true** means unconditional state shift.

We present a different phase decision function in this section. The derivation of the CR range is based on the optimum CR value. The optimum CR ($CR_{opt}$) is the value which is yielded by the optimum parallelism degree or together with its optimum thread mapping strategy. The upper and lower thresholds of the CR range are the optimum CR plus or minus a factor ($\delta$) of itself, as denoted in Equation 6. The initial value of $\delta$ is set to be 10% [2] and is later continuously modified. A low $\delta$ value can split a program into many short phases which is detrimental to performance, as the thread controllers over-react to environment changes. A high $\delta$ almost can never trigger the control actions. CR should not experience an abrupt jump while program still remains in the same phase. $\delta$ gradually rises at each false control action to diminish over-reaction of controllers. More specifically, $\delta$ increments 1% each time when the new predicted parallelism degree equals the previous value or the new value delivers worse performance, meaning that the sensors overreact to the CR fluctuation when the program still executes at the same phase. To ensure that the controller can be still responsive to phase changes, CR range stops expanding when $\delta$ reaches 15%. We address this CR range

[2]The initial value 10% is from the work of *Ansari et al.* [7]

decision function as *advanced phase decision function* in contrast with the simple phase decision function in Section 3.2.3.

$$CR = CR_{opt} \pm \delta * CR_{opt} \tag{6}$$

## 4.3 Coordination Policy

The cost of setting a thread mapping strategy can be high, therefore, the frequency of adjusting the strategy should be low. The frequency of switching mapping strategies brings non-trivial impacts on application performance, as the improvement of memory resource utilisation obtained by thread mapping can be forfeited by its potentially associated cost of thread migration. The thread mapping loop can only be invoked by the master loop under the following conditions:

- The predicted parallelism degree is no more than half of the core number. When the active thread number is equivalent to the core number, there is little interest to profile thread mapping strategies. This is due to the fact that the threads of our TM applications tend to behave similarly, *e.g.* the performance is not affected when two threads change their affinity. Furthermore, some TM applications experience significant performance degradation when their parallelism degrees increase, or their performance becomes unstable. Therefore, thread mapping is less interesting under such circumstances. When the parallelism degree surpasses this value, applications progress with the default mapping strategy (**Linux**).

- The new parallelism degree differs from its value before prediction. To further alleviate the cost of thread migration, yet ensure the controller to be responsive to phase changes, the mapping strategies are only re-profiled if the thread number fluctuates more than the core number that shared by two L2 caches. This is a tuning parameter which can be chosen to strengthen the responsiveness of control algorithms.

In addition, it is worth noting that the slave loop can be activated or disabled externally based on user requirements. In this case, the control loop becomes the *probabilistic model* described in Section 3.3 with the difference in the phase decision function.

## 4.4 The Slave loop for Thread Mapping Adaptation

The slave loop as illustrated in Fig. 10 performs the decision of optimum thread mapping strategy. It is invoked when certain conditions are satis-

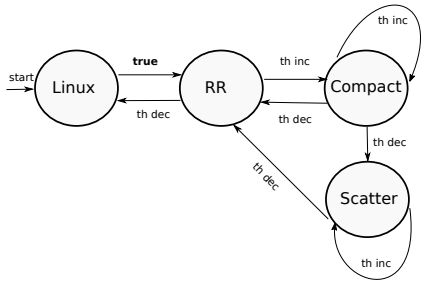



Figure 10: The automaton for thread mapping decision. *th* stands for throughput.

fied which have been detailed in Section 4.3. We obtain the optimum mapping strategy by selectively profiling the four strategies and comparing their throughputs. The automaton starts with *Linux* state where **Linux** strategy is profiled. Next, its state unconditionally shifts to *RR* state profiling **Round-Robin** strategy. **Round-Robin** is an intermediate strategy between **Compact** and **Scatter**, hence is profiled preceding to the other two. The automaton only shifts to the *Compact* state if the throughput yields by **Round-Robin** shows improvement contrasting with **Linux**, the automaton returns to *RR* state otherwise. The *Scatter* state is only invoked if **Compact** produces lower throughput than that of **Round-Robin**. The automaton remains in the *Compact* state when **Compact** outperforms **Round-Robin**. **Compact** and **Scatter** map the threads to cores in the opposite way making it less likely that both can outperform their intermediate strategy **Round-Robin**. In the end, **Scatter** is only selected as the optimum strategy when it gives higher throughput than that of **Round-Robin**. The state shifts back to *RR* state otherwise.

In the worst case, all the mapping strategies are profiled when **Scatter** is the optimum strategy. Whereas only **Linux** and **Round-Robin** are tried in the best case when **Linux** is the optimum.

# 5 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 synchronisation cost for information gathering but the obtained information may not represent the global view. Furthermore, the master thread must be active during the whole program execution possibly resulting in its termination prior to the other threads,

meaning that the fair execution time among threads can not be guaranteed. The latter method may suffer from synchronisation cost but the profile information represents the global view. More importantly, the latter method enables threads to be scheduled in a way that they can progress equally. We choose the second method. The synchronisation cost of information gathering is negligible for most of our applications. A thread is only suspended when it completes its current running transaction (a transaction commits). This can prevent a transaction from abolishing its completed tasks. All threads are awakened when reaching a thread barrier and threads are re-adjusted after exiting the barrier.

```
/*control functions*/
control_funcs(time,commits,aborts){
 ...
 adjust thread number;
 adjust thread mapping strategy;
 compute CR range;
 decide control decision frequency;
 ...
}
```

```
//entry point
stm_thread_init(){//tinystm sys call
 ...
 control_funcs();//suspend threads
 ...
}
```

```
 //entry point
 stm_thread_exit(){//tinystm func call
  ...
  control_funcs();//awake threads
  ...
 }
```

```
//entry point.
/*The info of all threads are*/
/ *synchronised at this entry point*/
stm_commit() {//tinystm sys call
  ...
 commit sensor;//collect commits
 time sensor; //record time
 ...
 control_funcs();// all the control
 ...
}
```

```
stm_abort(){//tinystm func call
 ...
 abort sensor;//collect aborts;
 ...
}
```

```
stm_init(){//tinystm func call
 ...
 time sensor;
 ...
}
```

Figure 11: The three entry points of the monitor and the control functions for TinySTM.

We have implemented a monitor to collect the profile information and control the dynamic parallelism. The monitor is a cross-thread lock which is composed of variables accessed by threads concurrently. The major variables of the monitor are commits, aborts, two FIFO queues recording the suspended and active threads, the best parallelism degree, the best mapping strategy and the throughput. More technical details are documented in [33]. Fig. 11 illustrates the designed monitor that incorporates three entry points. The first entry point is upon thread initialisation, where some initial values (*e.g.* thread id) are set for the threads. All the threads (in the *probabilistic model*) or only two pass (in the *simple model*) the first 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 awakened when one thread

exits. The monitor suspends and awakens threads by calling the *pthread* functions, *i.e. pthread_cond_wait()* and *pthread_cond_signal()*. Additionally, thread affinity is set up by Linux system function call *pthread_setaffinity_np()*. In control terminology, the monitor and the Linux function call are the effectors which carry out the decisions made by our autonomic manager. The control functions (*control_funcs()*) in Fig. 11 are partially programmed by Heptagon language.

A time overhead is added to each transaction when calling and releasing a monitor. The overhead caused by calling a lock is negligible on the transaction with medium and long length. Nevertheless this overhead is significant for the transaction with a small number of operations. This is the case for **intruder** and **ssca2** (two applications 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.

The *profile length* determines the frequency of control actions. The applications with the same magnitude of transactions share the same profile length. For instance, **genome** and **vacation** (two benchmarks from **STAMP** [34]) share the same profile length as the total number of transactions in the two applications are on the same magnitude (*i.e.* $10^6$).

To avoid thread starvation, we employ round-robin thread rotation to periodically awaken early suspended threads and suspend the running threads having executed longest time. Alas, this procedure brings thread migration which can be costly. To reduce its influence, a new awaken thread is mapped to the core where a thread is just suspended. The residual active threads keep their mapping in preference to migrating all the active threads whenever a thread changes its status.

# 6 Performance Evaluation

We present performance evaluation on six different **STAMP** [34] benchmarks and two applications from **EigenBench** [35]. **EigenBench** and **STAMP** are widely used for performance evaluation on TM systems. The data sets of our selected applications cover a wide range of parameters from short-length to long-length transactions, from short to long program execution time, from low to high program contention. Table 1 presents the qualitative summary of each application's online transactional characteristics: Tx (transaction) length or Tx size (the number of instructions per transaction), execution time, and contention (the global contention). The classification is based on the application executed with its static optimum parallelism on our

platform. A transaction with execution time between 10 $\mu s$ and 1000 $\mu s$ is classified as medium-length. The contention between 0.3 and 0.6 is classified as medium. The execution time between 10 seconds and 30 seconds is classified as medium.

| Application | Tx length | Execution time | Contention |
|---|---|---|---|
| **EigenBench** | medium | long | medium |
| **ssca2** | very short | short | low |
| **intruder** | short | medium | high |
| **genome** | medium | short | high |
| **vacation** | medium | medium | low |
| **yada** | medium | medium | high |
| **labyrinth** | long | long | low |

Table 1: Qualitative summary of each application's online transactional characteristics. The classification is based on the application execution with its optimum parallelism on our platform.

## 6.1 Platform

We evaluate the performance on a SMP machine with 4 processors of 6 cores each. Each core has a L1 cache (32KB). Every pair of cores share a L2 cache (3072KB) and every 6 cores share a L3 cache (16MB). This UMA machine holds 2.66GHz frequency and 64GB RAM. We utilise TinySTM as our STM platform.

## 6.2 Benchmark Settings

| | |
|---|---|
| loops | 16667 |
| A1 | 35536 |
| A2 | 1048576 |
| A3 | 8192 |
| R1 | 30 |
| W1 | 30 |
| R2 | 20 |
| W2 | 200 |
| R3i | 10 |
| W3i | 30 |
| R3o | 10 |
| W3o | 10 |
| NOPi | 0 |
| NOPo | 0 |
| Ki | 1 |
| Ko | 1 |
| LCT | 0 |

(a) one phase

| | | | | |
|---|---|---|---|---|
| | | *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 |

(b) two phases

Figure 12: Inputs of two EigenBench applications for 24 threads.

Two applications from **EigenBench** are evaluated, *i.e.* one application with one phase and one with two phases. They are selected and configured to serve as complementary benchmarks to **STAMP** for performance evaluation on special issues. **EigenBench**, with one phase, gives stable

online behaviour, hence it is ideal to demonstrate the overhead of control actions. Fig. 12(a) provides its input data set. The application with two phases is designed to verify if a change of parallelism requires a change of thread mapping strategies. Fig. 12(b) shows its inputs. This application is necessary, as most of the transactions in each **STAMP** application usually hold very similar behaviour [9] making them unsuitable for evaluation of the dynamic thread mapping strategies. **EigenBench** includes three different arrays which provide shared transactional access (Array1), private transactional access (Array2) and non-transactional access (Array3). Such a design enables us to easily tune the parameters (called data sets) of **EigenBench** to serve for different evaluation purposes. An approach to create several phases is to provide several input data sets and execute them in a sequence, as each data set can give individual behaviour.

We have evaluated 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 the paper. Since **bayes** exhibits non-determinism [36]: the ordering of commits among threads at the beginning of an execution can dramatically affect the execution time. The number of commits shows significant differences during each execution for **kmeans**, therefore, it is also excluded from performance evaluation. The inputs of the six applications are detailed in Fig. 13.

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

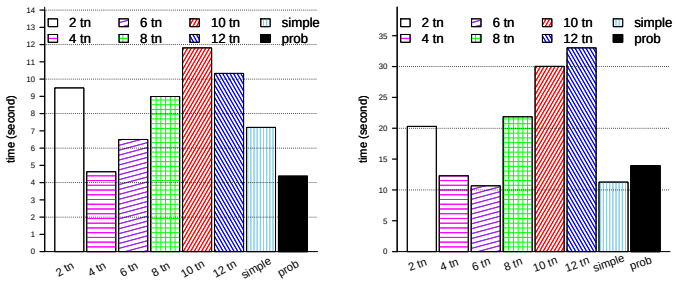
Figure 13: The inputs of **STAMP** applications.

## 6.3 Results

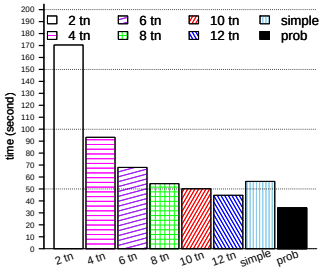
We firstly illustrate the performance evaluation for the *simple model* and *probabilistic model* introduced in Section 3. The results on coordination of parallelism and thread mapping from Section 4 are presented next. We set the maximum parallelism degree to be 24 which is the number of the available cores. We restricted the minimum parallelism degree to be 2, as we are only concerned with parallel applications. All the applications are executed 10 times and the results are the average execution time. Additionally, we implemented the *SimpleAdjust* algorithm proposed by *Ansari et al.* [7]. This algorithm starts the program with 8 threads and increments or decrements the parallelism degree by one when CR is beyond the range of 0.3 and 0.6. We presented the performance comparison with our models.

### 6.3.1 Parallelism Adaptation

Due to the page limit, we only illustrate the execution time difference between the different static parallelism degrees (up to 12) and adaptive parallelism for three selected applications (*i.e.* **genome**, **intruder** and **labyrinth**) in Fig. 14. The details of performance comparison are summarised in Table 2 and Table 3.




(a) **genome** (b) **intruder**




(c) **labyrinth**

Figure 14: Time comparison for selected applications on static and adaptive parallelism.

In the two tables, the results of both models are compared against the *best* (the static parallelism degree which gives the best performance), *average* (the average performance of the static parallelism degrees from 2 to 24) and *worst case* (the static parallelism degree which performs the worst) results of static parallelism. This comparison indicates that our models can outperform the static parallelism when an unknown application is considered. The performance of our models are also compared with the *SimpleAdjust*. The digits in the brackets are the static parallelism degrees which gives the best and the worst performance respectively. The symbol plus (+) means

performance gain against the compared value. It is worth noting that some of TM applications scale poorly when their parallelism rises (*e.g.* **genome**, **vacation** and **EigenBench**) due to TM mechanisms. For example, a highly contended phase results in continuous aborts from the same transactions, and thus slows down program progress. According to Table 2 and Table 3, our adaptive models outperform the majority of the static parallelism. The *probabilistic model* shows better performance on applications: **genome**, **vacation** and **labyrinth** against the *simple model*, but it indicates performance degradation on **yada** and **intruder** against the *simple model*. Both models present similar performance on **EigenBench** and **ssca2**. Our *simple model* shows significant performance loss on **genome** and **labyrinth**. **labyrinth** and a phase of **genome** perform the best with the maximum thread number and suffer from significant performance loss before reaching this value. *Ansari et al.* starts the applications with 8 threads while our *simple model* starts with the minimum. The performance penalty caused by the *simple model* on the two applications can be significantly diminished when starting the program with a higher thread number. The *probabilistic model* demonstrates overwhelming performance enhancement on this circumstance, as less time is spent in profiling the optimum thread number.

| benchmarks | best case | average | worst case | Ansari [7] |
|---|---|---|---|---|
| **EigenBench** (one phase) | -7% (12) | +10% | +50% (2) | +25% |
| **genome** | -57% (4) | +95% | +99% (20) | -30% |
| **vacation** | -45% (8) | +79% | +92% (24) | +39% |
| **labyrinth** | -52% (24) | +5% | +67% (2) | -42% |
| **yada** | -3% (8) | +66% | +91% (22) | +29% |
| **ssca2** | -14% (24) | +11% | +62% (2) | -9% |
| **intruder** | -6% (6) | +62% | +71% (24) | +14% |

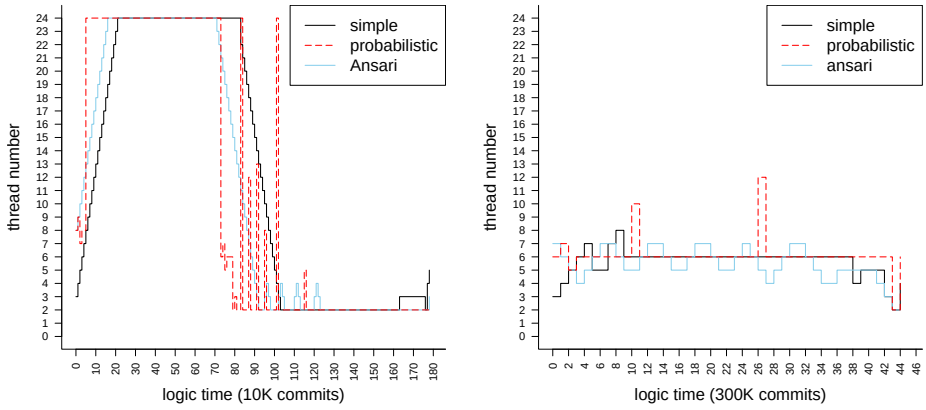
Table 2: Performance comparison with *simple model*. The higher value, the better performance. Plus (+) means performance gain against the compared value. The digits in the brackets are thread numbers.

| benchmarks | best case: | average | worst case | Ansari [7] |
|---|---|---|---|---|
| **EigenBench** (one phase) | -5% (12) | +11% | +51% (2) | +27% |
| **genome** | +3% (4) | +97% | +99% (20) | +21% |
| **vacation** | -18% (8) | +83% | +93% (24) | +50% |
| **labyrinth** | +8% (24) | +42% | +80% (2) | +14% |
| **yada** | -17% (8) | +61% | +90% (22) | +19% |
| **ssca2** | -16% (24) | +10% | +61% (2) | -10% |
| **intruder** | -31% (6) | +53% | +64% (24) | -7% |

Table 3: Performance comparison with *probabilistic model*.

Fig. 15 elucidates the online parallelism variation with *simple*, *probabilistic model* and model of *Ansari et al* for **genome** and **intruder**. The results given are based on one execution whose performance is the closest to the average execution time. As indicated in the figures, the *probabilistic model* is likely to give abrupt parallelism changes contrasting with the *simple model*, since the *probabilistic model* only requires one profile length

for parallelism prediction making it respond fast to CR changes. **genome** experiences three phases online. The first phase is short (two or three profile lengths) which contains both read and write operations. During the second phase, the transactions only include read operations resulting in $CR = 1$, hence the maximum parallelism is applied. The third phase is highly contended, hence low parallelism is given. As shown in Fig. 15(a), the *simple model* spends some time before reaching the optimum parallelism, thus some staircases reflect in the figure. The *probabilistic model* reacts fast to respond the CR and phase change, thus some abrupt parallelism changes are shown. Since less time is spent in reaching the optimum parallelism, the *probabilistic model* outperforms the *simple model* and Ansari's model on **genome**. *Per contra*, the *simple model* shows better performance than that of the *probabilistic model* on **intruder**. A sudden CR fluctuation is not always a sign of a new phase, therefore, the *probabilistic model* over-reacts to CR fluctuation resulting in performance loss. In contrast, the *simple model* avoids it.

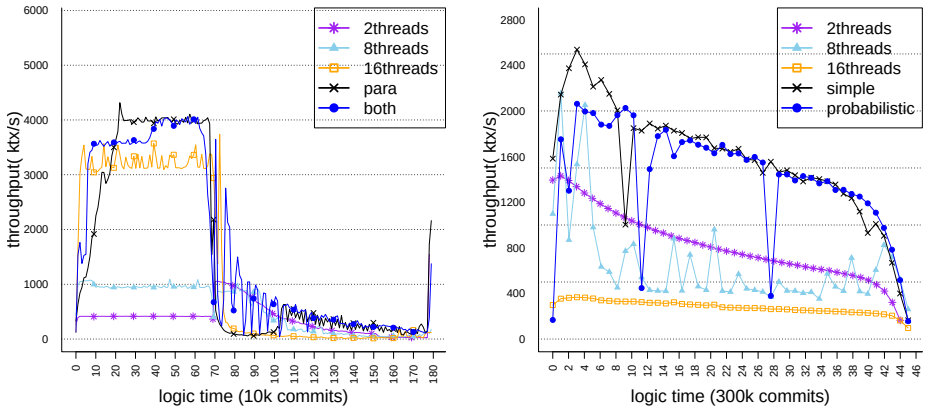



(a) **genome** (b) **intruder**

Figure 15: Online parallelism variation controlled by the *simple model* and *probabilistic model*.

The autonomic parallelism adaptation aims to regulate the parallelism which retains throughput at the optimum level at each phase. Ideally the throughput from the adaptive models should rival the one with the static parallelism which achieves the maximum throughput. We only present the online throughput changes of two applications **genome** and **intruder** here, more details are given in [33]. Fig. 16 elucidates their online throughput variation with static and adaptive parallelism. The black line with crosses

represents the throughput produced by the *simple model*, and the blue line with dots gives the throughput yielded by the *probabilistic model*.



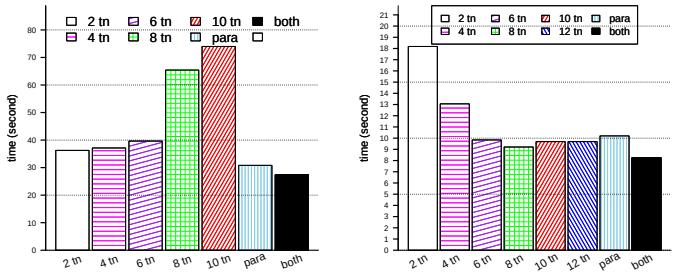

(a) **genome** (b) **intruder**

Figure 16: Online throughput variation of **genome** and **intruder**. The black line with crosses is the *simple model* and the blue line with dots is the *probabilistic model*.

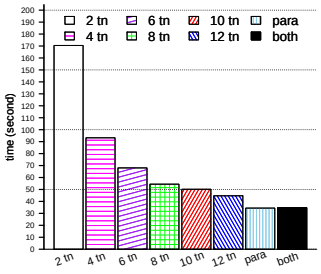
### 6.3.2 Parallelism and Thread Mapping Adaptation

In the following paragraphs, we address the model which only adjusts parallelism as *dynamic parallelism model*. We address the model that regulates both parallelism and thread mapping strategy as *dynamic thread control model*. We compare the performance of all the static parallelism with the two models. It is worth noting that despite the same parallelism prediction function, the *dynamic parallelism model* and *probabilistic model* (Section 6.3.1) differ in performance in certain cases, as their phase decision functions differ.

Fig. 17 illustrates the execution time comparison with different static parallelism and the two autonomic models for **EigenBench**, **yada** and **labyrinth**. The two models show the same performance on **labyrinth**. Thread mapping gives little impact on **labyrinth**, as its optimal parallelism degree equals the core number. Table 4 and Table 5 detail the performance comparison. According to the tables, our two models outperform the majority of the static parallelism. 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 models bring the similar performance to **labyrinth** and




(a) **EigenBench** (two phases) (b) **yada**




(c) **labyrinth**

Figure 17: Time comparison for selected applications on static and adaptive parallelism.

**ssca2**. Both thread models outperform the best case of static parallelism on **EigenBench** and **labyrinth**. Both thread models show 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. The *dynamic parallelism model* indicates better performance against the algorithm of *Ansari* for most of the applications while the *dynamic thread control model* outperforms it on all the applications. This means that thread mapping can further bring performance improvement.

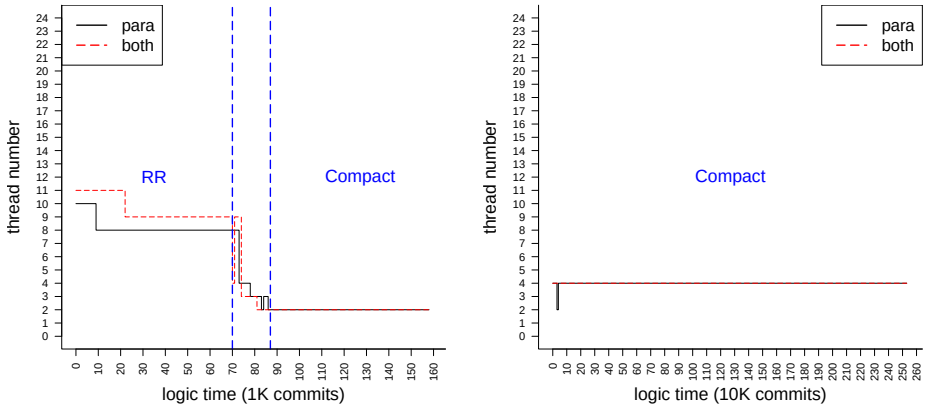
Fig. 18 demonstrates the online variation of parallelism degree for both models and the change of near-optimum thread mapping strategies controlled by the *dynamic thread control model*. In order to illustrate the performance impact of thread mapping strategies, the results presented in the two figures are the best results out of 10 executions. As shown in the figures, **EigenBench** shows variation of the optimum strategy, whereas **yada** keeps the same strategy during its whole execution since **yada** only incorporates one phase.

| benchmarks | best case | average | worst case | Ansari [7] |
|---|---|---|---|---|
| **EigenBench** (two phases) | +14% (2) | +95% | +99% (24) | -2% |
| **genome** | +16% (4) | +97% | +99% (20) | +29% |
| **vacation** | -13% (8) | +83% | +94% (24) | +52% |
| **labyrinth** | +7% (24) | +42% | +80% (2) | +14% |
| **yada** | -11% (8) | +63% | +91% (22) | +23% |
| **ssca2** | -4% (24) | +19% | +65% (2) | +1% |
| **intruder** | -48% (6) | +46% | +59% (24) | -20% |

Table 4: Performance comparison with *dynamic parallelism model*. The higher value, the better performance. Plus (+) means performance gain against the compared value. The digits in the brackets are thread numbers.

| benchmarks | best case | average | worst case | Ansari [7] |
|---|---|---|---|---|
| **EigenBench** (two phases) | +24% (2) | +96% | +99% (24) | +9% |
| **genome** | +0% (4) | +97% | +99% (20) | +12% |
| **vacation** | -26% (8) | +81% | +93% (24) | +47% |
| **labyrinth** | +6% (24) | +41% | +80% (2) | +14% |
| **yada** | +10% (8) | +70% | +93% (22) | +38% |
| **ssca2** | -4% (24) | +19% | +65% (2) | +1% |
| **intruder** | -9% (6) | +61% | +70% (24) | +11% |

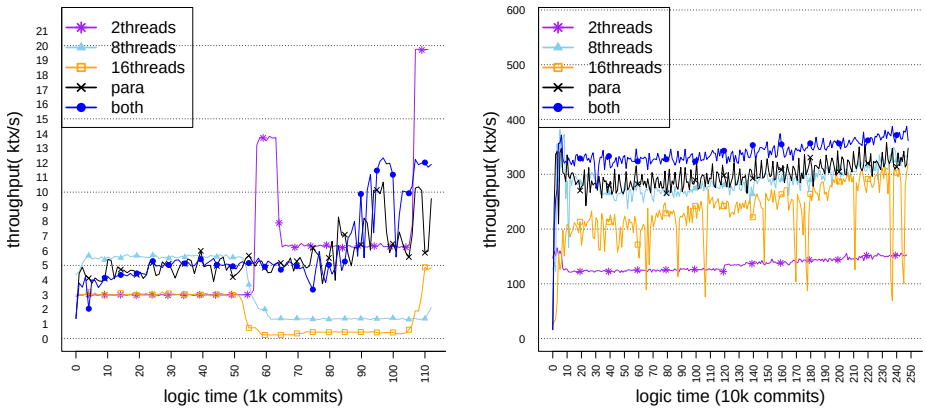
Table 5: Performance comparison with *dynamic thread control model*.




(a) **EigenBench** (two phases) (b) **yada**

Figure 18: Online parallelism variation by the two models for **EigenBench** and **yada**.

Lastly, Fig. 19 presents the comparison of online throughput variation for two applications. The *dynamic parallelism model* rivals the maximum throughput of the static parallelism at each phase. The *dynamic thread control model* can exceed the best throughput at certain phases. Note that, the static parallelism degrees are applied with the default mapping strategy **Linux**.




(a) **EigenBench** (two phases) (b) **yada**

Figure 19: Online throughput variation of **EigenBench** and **yada**. The black line with crosses is the *dynamic parallelism model* and the blue line with dots is the *dynamic thread control model*.

# 7 Discussion

The autonomic managers described in this paper are partially implemented in Heptagon, as Heptagon provides a straightforward way of programming automata. Moreover, automata can better elucidate the relations among the control decision functions. Nevertheless, our automata can be otherwise written by other programming languages (*e.g.* C\C++). We evaluate our approaches on **STAMP** and **EigenBench**, nevertheless, the *probabilistic model* can be applied to the TM applications whose threads may change their behaviour during their lifetime but show similar behaviour at each time slot. The *simple model* can extend to TM applications with heterogeneous threads, however, it performs worse than the *probabilistic model* when the optimum parallelism degree of the applications is high.

To avoid thread starvation, we employ round-robin thread rotation to periodically awaken early suspended threads and suspend the running threads having executed longest time. Round-robin thread rotation is necessary for applications with a set amount of transactions allocated to each thread. This is the case for **EigenBench**, without thread rotation the suspended threads starve, which not only slows down its execution, but also changes its online behaviour. In some applications such as **yada**, the transactions are allocated to each thread dynamically, and are not assigned to the suspended threads, hence applications likewise require no round-robin thread rotation.

Table 6 lists the effect of execution time on diverse applications stemming from round-robin thread rotation.

| Application | Effect | Applications | Effect | Application | Effect | Application | Effect |
|---|---|---|---|---|---|---|---|
| **EigenBench** (one phase) | yes | **intruder** | no | **genome** | no | **vacation** | yes |
| **EigenBench** (two phases) | yes | **ssca2** | no | **labyrinth** | no | **yada** | no |

Table 6: The effect of round-robin thread rotation on applications.

**Compact** is favoured when CR is low and **Scatter** is likely to be selected when CR is high. When CR is high, the *dynamic thread control model* favours higher parallelism. Some mapping strategies do not bring a significant performance difference for certain applications, therefore, the decision of optimum mapping strategies can differ at each execution. When the parallelism is approaching the maximum core number, the thread mapping profiling is disabled as little performance impact can be received from thread mapping.

## 7.1 Overhead Analysis

The overhead of our approaches mainly originate from two aspects which are caused by the effectors of the feedback control loops.

*Thread migration.* This stems from two points: switching among the four mapping strategies and periodically awaking or suspending the threads to ensure their fair execution time. A good mapping strategy can lead to decrement of cache misses, therefore, overcomes performance degradation caused by re-mapping threads. For example, as we have demonstrated that the *dynamic thread control model* outperforms the *dynamic parallelism model* on **yada**. The former model gives 19% fewer cache misses to the latter one, but causes 6% more thread migration. Nevertheless, frequent thread migration may forfeit the performance gain obtained by lower cache misses.

*The cost of calling the monitor.* Two factors contribute to this cost: the operation of requesting and releasing the monitor and the time spent in waiting for the monitor. The latter cost increases significantly with higher active number of threads and gives significant impact on the applications with short-length transactions. However, this cost is trivial, more specifically it is less than 2% for the transaction with medium length and long length. This overhead is higher for the short-length transaction, however, it is diminished through lower calling frequency of the monitor as described in Section 5.

## 7.2 Limitations

Performance penalty can occur when program parallelism varies, yet is trivial on shared memory. To further reduce the penalty, a thread is only suspended

when its current transaction commits (Section 5). We have described two phase decision functions to compute CR ranges to trigger or disable control actions. The simple phase decision function shows its limitations on phase detection for **vacation**. Since its CR tends to fluctuate frequently over the CR range, yet the program remains at the same phase. Therefore, both the *simple model* and *probabilistic model* overreact to such changes and perform unstably. **vacation** suffers from the same performance loss with the advanced phase decision function. Its performance is further forfeited with mapping strategy profiling.

The parallelism decision function are based on ideal situations, thus the predicted parallelism *de facto* may be sub-optimum for certain applications (*e.g.* **yada**). However, such performance loss can be compensated by the performance gain from thread mapping. In general, 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 salient behaviour variation online. It is also worth noting that thread mapping is not necessary for all the applications, *e.g.* the application (*e.g.* **labyrinth**, **ssca2**) with low contention and its parallelism degree equals the core number. Therefore, both *dynamic parallelism model* and *dynamic thread control model* illustrate similar performance on the two applications. Additionally, profiling the mapping strategy gives penalty to **genome** and **vacation**, as **genome** contains occasional sudden variation of contention and contention of **vacation** fluctuates frequently. A refined phase decision function can improve the performance of **vacation**. The *dynamic parallelism model* can respond immediately. In contrast, the *dynamic thread control model* requires extra profiling lengths to search a better mapping strategy, meanwhile the applications have already entered a new phase.

Our instrumentation adds a time stall at commit time. This makes the contention management policy act similar as a *backoff* policy (except the *backoff* policy gives proportional time stall based on the retry times). Therefore, some applications show slightly better or worse performance than the ones without instrumentation.

Due to the impact of thread migration between switch of mapping strategies, the predicted strategy can be sub-optimum. In this work, we focus on performance evaluation on UMA. Our mapping strategies do not take into account of the NUMA [9]. Furthermore, with the influence of memory location and thread migration, our parallelism models do not scale well on NUMA systems. The performance on NUMA has been illustrated in our work in [33]. The possible solutions would be (1) implementing new mapping strategies which takes into account of the distributed memory on NUMA (2)

and setting memory affinity [37, 38].

The metrics (*i.e. commits, aborts*) that serve as inputs to the feedback control loops are TM-specific, therefore, our approaches are not applicable to the non-TM platforms. However, it is possible to apply our approaches to non-TM platforms, if we substitute the CR and throughput with thread contention and instruction throughput. We leave this for future work.

# 8 Related Work

## 8.1 Parallelism Adaptation

State-of-the-art approaches, which address parallelism adaptation for TM or non-TM systems, can be classified into two categories: with and without feedback control loops.

### 8.1.1 Parallelism adaptation without feedback control

*Raman et al.* [39] presented an a library which is built on top of Pthreads library to facilitate the writing of parallel programs and improve their performance (by adjusting task scheduling and thread parallelism degrees) when their working environments differ. Its mechanisms of online parallelism adaptation requires a developer to modify programs in order to specify certain environment features. We provide an API that does not require such instrumentation to source code.

DOPE [39] and Parcae [40] were compiler-based approaches for dynamic parallelism adaptation. Both approaches paused program execution while altering its parallelism degree. Our two methods for parallelism control perform without halting program execution.

### 8.1.2 Parallelism adaptation with feedback control

Feedback control loops have been adopted as cornerstones of software-intensive and self-adaptive systems [27]. It has been been addressed in previous works [11, 7, 12, 13, 14, 15, 16, 17, 18, 19, 41, 42, 43, 44] to dynamically adapt thread parallelism using control techniques for both TM and non-TM systems. These works either require offline training procedures to obtain an initial form of a function for parallelism prediction, or demand to increment/decrement the thread number progressively in order to search its optimal value. A notable exception in [45] described a Markov chain-based model which is able to perform parallelism prediction relying on sampling four online metrics. Both of our models on parallelism control only sample two metrics. Fewer sampling metrics cause less online overhead.

*Heiss et al.* [18] proposed two approaches to dynamically adjust thread number: (1)adjusting by one until performance becomes worse and (2) determining the number of thread number to regulate by a more sophisticated control function called *Parabola Approximation* (PA). The coefficients of PA function are continuously improved based on the throughput. Similarly, *Ansari et al.* [11, 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 pre-set CR range or is not equal to a single predefined CR value. *Ansari et al.* gave five different algorithms that searched the best parallelism degree linearly or exponentially. Likewise, *Chan et al.* [41] presented two approaches which simply increase and decrease the parallelism degree by one based on the CR and commit rate (the number of commit per unit time). *Ravichandran et al.* [15] presented a model which adapts the thread number in two phases: *exponential* and *linear*. Similarly, in [42] the parallelism degree is incremented cubically and later regulated linearly. During decrement, linear decrements are followed by multiplicative decrements. Our parallelism control models can resolve the CR range which is adaptive to the online program behaviour rather than a fixed value as in [11, 7]. Additionally to the *simple model* that searches the optimal parallelism degree, we presented a *probabilistic model* which does not require the linear or exponential search phases, thus it can shorten the profiling time.

*Rughetti et al.* [12, 13] utilised a neural network (machine-learning approach) to enable 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. In [16], *Rughetti et al.* integrated the machine learning approach with and analytic approach, which not only reduced the number of training samples but also improved the prediction precision than that of a pure analytic model. A similar work from *Di Sanzo et al.* [17] proposed to estimate the transaction abort probability rather than the wasted time. Nevertheless their approach of concurrency regulation is limited to the optimistic concurrency control where transactions are aborted and resumed right upon conflicts. *Didona et al.* [14] 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. Instead of using machine learning techniques, in [19], the parallelism degree was predicted by a Collaborative Filtering (CF) prediction technique. CF could estimate a preferred parallelism degree through exploiting preferences and ratings by the user from a set of training data. Analogising with above approaches, our methods do not need offline training

procedure to form prediction functions.

Some works have addressed parallelism adaptation on non-TM platforms. *Sridharan et al.* [43] proposed to linearly search the optimal parallelism degree. A program progresses with a non-optimal degree during the search period and it is likely that the program may already enter a new phase during the search. We proposed a *probabilistic model* which skips the search procedure and can compute a near-optimal parallelism degree. *Sridharan et al.* gave another approach in [44]. However, this approach requires to reset the parallelism degree to be 1 and to resume a new prediction. Resetting parallelism degree can slow down program execution. Both of our approaches on parallelism adaptation do not perform parallelism reset.

In [46] we have provided the *simple* and *probabilistic* models to regulate the parallelism degree. A weak point of the two models is that their *phase decision functions* either require additional profile lengths to make decisions or are not sensitive enough to phase changes under certain conditions. This work presented an *advanced phase decision function* that is more adaptive and sensitive to phase changes. Moreover, this work enhances application performance with thread mapping control. The thread mapping control is served as a slave loop to the main loop: parallelism control loop.

## 8.2 Parallelism and Thread Mapping Adaptation

Some works targeted for non-TM systems. *Wang et al.* [47] gave 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 task mapping rules respectively. *Tournavitis et al.* [48] enhanced the work of *Wang et al.* by adding a profile-driven approach to the static compiler analysis. Without a compiler assistance, *Emani et al.* [8] utilised a number of different offline trained experts to perform prediction. However, the above works only dealt with task mapping rather than mapping threads to cores. Moreover importantly, our work does not require offline training.

*Diener et al.* [49] described two methods, *i.e. Exhaustive Search* and *Heuristic Algorithm*, to resolve thread placement issues. The authors proposed a data sharing metric used to measure how much a certain thread placement can benefit from data sharing. This is calculated by aggregating the shared cache access between two threads with a high metric indicating high data sharing among caches. Both methods require to perform simulation of the applications in order to obtain data sharing metrics. Contrasting with the above offline approaches, our feedback control loops regulate both

parallelism and thread mapping strategies online without offline simulation for STM systems.

One previous work from *Castro* [9] investigated the online thread mapping on STM applications. *Castro* utilised an offline training procedure to obtain a predictor of the optimum thread mapping strategy and employed it to estimate a thread-to-core mapping solution online. No prior literature has addressed the issue on coordination of parallelism and thread mapping for TM systems, with an exception of our work in [32]. In that work, the thread mapping decision function is only called from a single state of its automaton where all the mapping strategies are profiled. Whereas in this work, the thread mapping decision function forms a complete automaton which is controlled by a master loop. Furthermore we selectively profile the mapping strategies which reduce the performance penalty. Additionally, comparing with our work in [32], we designed a better phase decision function which can better prevent controller over-react to CR fluctuations.

### 8.3 Coordination of Feedback Control Loops

*Gueye et al* [50, 51] and *Rutten et al* [26] proposed the concepts of coordination of feedback control loops. *Gueye et al* designed two feedback control loops to adapt the frequency of CPU and the number of cluster nodes, respectively. An additional control loop is introduced to the system which coordinates the control actions of the former two loops. The three control loops function in parallel. The coordination loop delays the control actions of the loop which controls the number of cluster nodes until the maximum point of frequency has reached. In this case, the loop which controls the cluster node commences to add nodes.

Likewise, we adopt the concept of control loop coordination with one loop working as a master loop and one as a slave loop. However, our control loops serve with distinguishing control objectives and demonstrate significant differences in the loop designs.

## 9 Conclusion and Future Work

In this paper, we investigated autonomic management of thread parallelism and mapping on a STM system. We examined the performance of different static parallelism and concluded that online regulation of parallelism and thread mapping is necessary to the TM application performance. Next, we presented our approaches, *i.e.* utilising feedback control loops to automatically manage threads online. Their performance was then compared with

static parallelism. Lastly, we analysed the implementation overhead and discussed the advantages as well as limitations of our work.

Thread migration impacts system performance and causes performance degradation. Four different thread mapping strategies are selectively profiled in order to obtain the optimum one. Such a profiling procedure is relatively costly, as it brings thread migration and also imposes the program to work partly under unsuitable mapping strategies. We plan to design a thread mapping strategy predictor which can predict the optimum thread mapping strategy in one step. Compilers are able to analyse and provide sufficient information (*e.g.* the memory-intensive and computation-intensive code) of applications as well as their underlying hardware. This information together with online profile information can construct the predictor for the optimum strategy. Furthermore, our approach on thread control for STM system can be transferred to HTM or Hybrid systems which can demonstrate better performance by taking advantage of hardware support in future work.

# References


[1] Larus J, Kozyrakis C. Transactional memory. *Commun. ACM* Jul 2008; **51**(7):80–88.

[2] He Z, Yu X, Hong B. Profiling-based adaptive contention management for software transactional memory. *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, 2012; 1204–1215.

[3] Taubenfeld G. *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 2006.

[4] Herlihy M, Moss JEB. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News* May 1993; **21**(2):289–300.

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

[6] Damron P, Fedorova A, Lev Y, Luchangco V, Moir M, Nussbaum D. Hybrid transactional memory. *SIGPLAN Not.* Oct 2006; **41**(11):336–346.

[7] Ansari M, Kotselidis C, Jarvis K, Luján M, Kirkham C, Watson I. Adaptive concurrency control for transactional memory. *MULTIPROG '08: First Workshop on Programmability Issues for Multi-Core Computers*, 2008.

[8] Emani MK, O'Boyle M. Celebrating diversity: A mixture of experts approach for runtime mapping in dynamic environments. *SIGPLAN Not.* Jun 2015; **50**(6):499–508.

[9] Castro MB. Improving the performance of transactional memory applications on multicores: A machine learning-based approach. PhD Thesis, University de Grenoble December 2012.

[10] Kephart JO, Chess DM. The vision of autonomic computing. *Computer* Jan 2003; **36**(1):41–50.

[11] Ansari M, Kotselidis C, Jarvis K, Luján M, Kirkham C, Watson I. Advanced concurrency control for transactional memory using transaction commit rate. *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, Springer-Verlag: Berlin, Heidelberg, 2008; 719–728.

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

[13] Rughetti D, Sanzo PD, Ciciani B, Quaglia F. Dynamic feature selection for machine-learning based concurrency regulation in stm. *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, 2014; 68–75.

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

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

[16] Rughetti D, Sanzo PD, Ciciani B, Quaglia F. Analytical/ML Mixed Approach for Concurrency Regulation in Software Transactional Memory.

*2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014; 81–91.

[17] Sanzo PD, Re FD, Rughetti D, Ciciani B, Quaglia F. Regulating concurrency in software transactional memory: An effective model-based approach. *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, 2013; 31–40.

[18] Heiss HU, Wagner R. Adaptive load control in transaction processing systems. *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1991; 47–54.

[19] Didona D, Diegues N, Kermarrec AM, Guerraoui R, Neves R, Romano P. Proteustm: Abstraction meets performance in transactional memory. *SIGOPS Oper. Syst. Rev.* Mar 2016; **50**(2):757–771.

[20] Yoo RM, Lee HHS. Adaptive transaction scheduling for transactional memory systems. *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, ACM: New York, NY, USA, 2008; 169–178.

[21] McVoy L, Staelin C. Lmbench: Portable Tools for Performance Analysis. *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, USENIX Association: Berkeley, CA, USA, 1996; 23–23.

[22] Castro M, Goes LFWG, Mehaut JF. Adaptive thread mapping strategies for transactional memory applications. *Journal of Parallel and Distributed Computing* 2014; **74**(9):2845 – 2859.

[23] Agawal A, Gupta A. Memory-reference Characteristics of Multiprocessor Applications Under MACH. *SIGMETRICS Perform. Eval. Rev.* May 1988; **16**(1):215–225.

[24] Thekkath R, Eggers SJ. Impact of sharing-based thread placement on multithreaded architectures. *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*, 1994; 176–186.

[25] Huebscher MC, McCann JA. A survey of autonomic computing — degrees, models, and applications. *ACM Comput. Surv.* Aug 2008; **40**(3):7:1–7:28.

[26] Rutten E, Marchand N, Simon D. Feedback Control as MAPE-K loop in Autonomic Computing. *Software Engineering for Self-Adaptive Systems*. Lecture Notes in Computer Science, Springer, 2016.

[27] Litoiu M, Shaw M, Tamura G, Villegas NM, Müller H, Giese H, Rutten E, Rouvoy R. What Can Control Theory Teach Us About Assurances in Self-Adaptive Software Systems? *Software Engineering for Self-Adaptive Systems 3: Assurances*, de Lemos R, Garlan D, Ghezzi C, Giese H (eds.). Springer, 2016.

[28] Berekmeri M, Serrano D, Bouchenak S, Marchand N, Robu B. Feedback Autonomic Provisioning for Guaranteeing Performance in MapReduce Systems. *IEEE Transactions on Cloud Computing* 2016; **PP**(99):1–1.

[29] Cerf S, Berekmeri M, Marchand N, Bouchenak S, Robu B. Adaptive Modelling and Control in Distributed Systems. *Phd Forum 34th International Symposium on Reliable Distributed Systems (SRDS) 2015*, McGill University: Montreal, Canada, 2015.

[30] Delaval G, De Palma N, Gueye SMK, Marchand H, Rutten É. Discrete Control of Computing Systems Administration: A Programming Language Supported Approach. *European Control Conference*, Morari M (ed.), Zurich, Switzerland, 2013; 117–124.

[31] Delaval G, Marchand H, Rutten E. Contracts for Modular Discrete Controller Synthesis. *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2010)*, Stockholm, Sweden, 2010.

[32] Zhou N, Delaval G, Robu B, Rutten É, Méhaut JF. Autonomic Parallelism and Thread Mapping Control on Software Transactional Memory. *ICAC 2016 - 13th IEEE International Conference on Autonomic Computing* , Wursburg, Germany, 2016.

[33] Zhou N. Autonomic Thread Parallelism and Mapping Control for Software Transactional Memory. PhD thesis, UJF Grenoble-1 ; INRIA Grenoble Oct 2016.

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

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

[36] Ruan W, Liu Y, Spear M. STAMP need not be considered harmful. *9th ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, 2014.

[37] Antony J, Janes PP, Rendell AP. Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/Hypertransport. *Proceedings of the 13th International Conference on High Performance Computing*, HiPC'06, Springer-Verlag: Berlin, Heidelberg, 2006; 338–352.

[38] Wang W, Davidson JW, Soffa ML. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines. *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016; 419–431.

[39] Raman A, Kim H, Oh T, Lee JW, August DI. Parallelism Orchestration Using DoPE: The Degree of Parallelism Executive. *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, ACM: New York, NY, USA, 2011; 26–37.

[40] Raman A, Zaks A, Lee JW, August DI. Parcae: A system for flexible parallel execution. *SIGPLAN Not.* Jun 2012; **47**(6):133–144.

[41] Kinson Chan CLW King Tin Lam. Aptive thread scheduling techniques for improving scalability of software transactional memory,. *IASTED International Conference on Parallel and Distributed Computing Networks*, 2011.

[42] Mohtasham A, ao Barreto J. RUBIC: Online Parallelism Tuning for Co-located Transactional Memory Applications. *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, 2016.

[43] Sridharan S, Gupta G, Sohi GS. Holistic run-time parallelism management for time and energy efficiency. *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, ACM: New York, NY, USA, 2013; 337–348.

[44] Sridharan S, Gupta G, Sohi GS. Adaptive, efficient, parallel execution of parallel programs. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, ACM: New York, NY, USA, 2014; 169–180.

[45] Sanzo PD, Sannicandro M, Ciciani B, Quaglia F. Markov Chain-Based Adaptive Scheduling in Software Transactional Memory. *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016; 373–382.

[46] Zhou N, Delaval G, Robu B, Rutten É, Méhaut JF. Control of autonomic parallelism adaptation on software transactional memory. *2016 International Conference on High Performance Computing Simulation (HPCS)*, 2016; 180–187.

[47] Wang Z, O'Boyle MF. Mapping parallelism to multi-cores: A machine learning based approach. *SIGPLAN Not.* Feb 2009; **44**(4):75–84.

[48] Tournavitis G, Wang Z, Franke B, O'Boyle MF. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. *SIGPLAN Not.* Jun 2009; **44**(6):177–187.

[49] Diener M, Madruga F, Rodrigues E, Alves M, Schneider J, Navaux P, Heiss HU. Evaluating thread placement based on memory access patterns for multi-core processors. *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, 2010; 491–496.

[50] Gueye SMK, Palm ND, Rutten É, Tchana A, Berthier N. Coordinating self-sizing and self-repair managers for multi-tier systems. *Future Generation Computer Systems* Jun 2014; **35**:14 – 26.

[51] Gueye SMK, de Palma N, Rutten É. Coordination control of component-based autonomic administration loops. *15th International Conference on Coordination Models and Languages, COORDINATION*, Florence, Italy, 2013.