HAL
archives-ouvertes.fr

Control of Autonomic Parallelism Adaptation on Software Transactional Memory

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

▶ **To cite this version:**

Naweiluo Zhou, Gwenaël Delaval, Bogdan Robu, Eric Rutten, Jean-François Méhaut. Control of Autonomic Parallelism Adaptation on Software Transactional Memory. International Conference on High Performance Computing & Simulation (HPCS 2016) , Jul 2016, Innsbruck, Austria. pp.180-187, 2016, .

HAL Id: hal-01309195
https://hal.archives-ouvertes.fr/hal-01309195

Submitted on 21 Nov 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Control of Autonomic Parallelism Adaptation on Software Transactional Memory

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

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

***Abstract*—Parallel programs need to manage the trade-off between the time spent in synchronization and computation. A high parallelism may decrease computing time while increase synchronization cost among threads. A way to improve program performance is to adjust parallelism to balance conflicts among threads. However, there is no universal rule to decide the best parallelism for a program from an offline view. Furthermore, an offline tuning is error-prone. Hence, it becomes necessary to adopt a dynamic tuning-configuration strategy to better manage a STM system. Software Transactional Memory (STM) has emerged as a promising technique, which bypasses locks, to address synchronization issues through transactions. Autonomic computing offers designers a framework of methods and techniques to build automated systems with well-mastered behaviours. Its key idea is to implement feedback control loops to design safe, efficient and predictable controllers, which enable monitoring and adjusting controlled systems dynamically while keeping overhead low. We propose to design feedback control loops to automate the choice of parallelism level at runtime to diminish program execution time.**

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

I. Introduction

Multicore processors are ubiquitous, which enhance program performance through thread parallelism (number of simultaneous active threads). Although a high parallelism degree shortens execution time, it may also potentially increase synchronization time. Therefore, it is crucial to find the trade-off between synchronization and computation. The conventional way to address synchronization issues is via locks. However, locks are notorious for various issues such as the likelihood of deadlock and the vulnerability to failure and faults. Additionally it is not straightforward to analyse interactions among concurrent operations.

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

Online parallelism adaptation began to receive attention recently in TM systems. The level of a suitable parallelism in a program can significantly affect system performance. However it is onerous to set a suitable parallelism degree for a program offline especially for the one with online behaviour variation. When apropos of the program with online behaviour fluctuation, there is no unique thread number can enable its optimum performance. Therefore, the natural solution is to monitor a program at runtime and alter its parallelism when necessary.

We introduce autonomic computing [4] to STM systems to automatically regulate online program parallelism. In this paper we argue that online adaptation is necessary and feasible for parallelism management in STM systems. We demonstrate that the program performance is sensitive to the parallelism. We present two effective profiling frameworks for the parallelism adaptation on TinySTM [2]. The main contributions of our paper are as follows:

1) We present two adaptive profiling frameworks that detect the suitable parallelism degree in order to optimize system performance at runtime.
2) We dynamically resolve a phase detection method.

The rest of the paper is organized as follows. Section II summaries the background and related work. Section III details the profiling procedures and the online parallelism adaptation methods. Section IV presents the implementation details. Section V shows the results. Section VI discusses the pros and cons of our adaptive framework and Section VII concludes the paper and gives future work.

II. Background and Related Work

A. Background on Software Transactional Memory

Transactional memory (TM) is an alternative synchronization technique. Its accesses to shared data are enclosed in transactions which are executed speculatively without being blocked by locks. Each transaction makes a sequence of tentative changes to shared memory. When a transaction completes, it can either *commit* making the changes permanent to memory

or *abort* hence discarding the previous changes [1]. Two parameters are often used in TM to indicate system performance, namely *commit ratio* and *throughput*. Commit ratio (CR) equals the number of commits divided by the sum of number of commits and number of aborts; it measures the level of conflict or contention among the current transactions. Throughput is the number of commits in one unit of time; it directly indicates program performance. TM can be implemented in software, hardware or hybrid. Different mechanisms explore the trade-off that impact on performance, programmability and flexibility. In this paper, we focus on STM systems and utilise TinySTM [2] as our experimental platform. TinySTM is a lightweight STM system which adopts a shared array of locks to control the concurrent accesses to memory and applies a shared counter as its clock to manage transaction conflicts.

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

B. Background on Autonomic Computing

Autonomic computing [5] is a concept that brings together many fields of computing with the purpose of creating computing systems that self-manage. A system is regarded as an autonomic system if it supports one of the following features [4]: (1) **self-optimization**, the system seeks to improve its performance and efficiency on its own; (2) **self-configuration**, when a new component is introduced into a system, the component is able to learn the system configuration. (3) **self-healing**, the system is able to recover from failures; (4) **self-protection**, the system can defend against attacks.

In this paper, we concentrate on the first feature: self-optimization. We introduce feedback control loops to achieve autonomic parallelism management. A classic feedback control loop is illustrated in Fig. 1 in the shape of a MAPE-K loop [4].

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

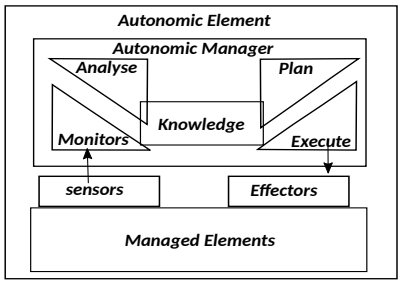


Fig. 1. **A MAPE-K control loop. It incorporates an autonomic manager, a sensor, an effector as well as a managed element among which the autonomic manager plays the main role.**

C. Related Work

It has been been addressed in previous work [6], [7], [8], [9] to dynamically adapt parallelism degrees via control techniques to reduce wasted work in TM systems. *Ansari et. al.* [6] proposed to adapt parallelism online by detecting the changes of the application's CR. The regulation action is made to the parallelism if the CR falls out of the pre-set CR range or is not equal to a single preset CR value. This is based on the fact that CR falls during highly contended phases rises with low contended phases. *Ansari et. al.* gave five different algorithms which decide the profile length and the level of parallelism. *Ravichandran et. al.* [7] presented a model which adapts the thread number in two phases: *exponential* and *linear* with a feedback control loop. *Rughetti et. al.* [9] utilise a neural network to estimate performance of STM applications. The neural network is trained to predict execution time of wasted transactions which in turn is utilised by a control algorithm to regulate the parallelism. *Didona et. al.* [8] introduces 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.

Our approaches differ from the previous work as (1) comparing with *Ansari et. al.*, we resolve a CR range for phase detection which is adaptive to the online program behaviour rather than a fixed range or a single preset value; (2) analogising with the parallelism prediction by *Ravichandran et. al.*, *Didona et. al.* and *Rughetti et. al.*, we present a model which predicts the optimum parallelism based on probability theory which requires no offline training procedure or to try different thread number to search the optimum; (3) contrasting with the aforementioned work that either only use CR or throughput to indicate program performance, we employ both, *i.e.* CR to indicate the program phase and throughout to indicate correctness of the parallelism adaptation.

III. Autonomic Parallelism Adaptation

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

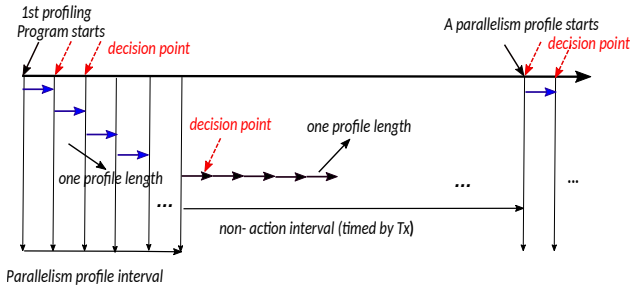


Fig. 2. **Periodical profiling procedure. At each decision point (marked by dashed red arrow), the actions are taken. Each decision points corresponds to one state of an automaton as in Fig. 3(b) and Fig. 5.**

We measure three parameters from the STM system, namely the number of commits, the number of aborts and physical time. The number of commits and the number of aborts are addressed as commits and aborts subsequently. We choose CR and throughput to denote program performance, as CR and throughput are both sensitive to parallelism variation. Either is by itself not sufficient enough 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 simply just a low number of transactions taking places.
- CR indicates the conflicts among threads. A high CR means low synchronization time whereas a low CR mean high synchronization cost. But a low CR can bring a high throughput when a large number of transactions are executing concurrently, whereas a high CR may give low throughput due to a small number of transactions executing concurrently.

The controller observes 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.

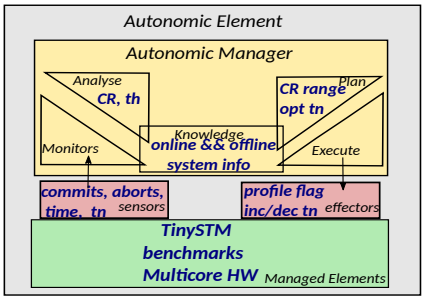
A. Overview of Profiling Algorithm on simple model

To describe the simple model, we firstly give an overview of the profiling procedure and later describe the model through the prism of control theory.

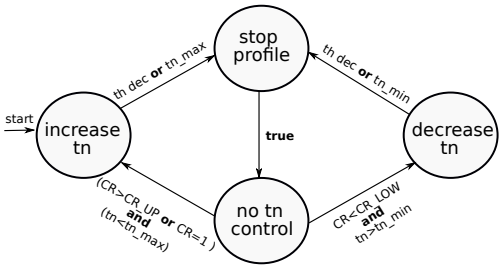
To achieve autonomic parallelism adaptation which provides a program with its optimum parallelism, we propose to periodically profile the parallelism and select the value that achieves the highest throughput. By observing CR, we can obtain the contention information of an application. CR usually fluctuates in a certain range within the same phase. When a program enters a new phase, the current parallelism produces a different CR which falls out of the current CR range. The CR fluctuation triggers a new parallelism adaptation action. Initially the two CR thresholds (upper CR and lower CR thresholds) are both set to be 0 and are trained in the later profile stage. The detail of the profiling procedure is illustrated in Fig. 2.

The parallelism profiling procedure starts once the program starts. Initially the program creates a pool of threads, among which only 2 threads are awaken and the rest are suspended. At each decision point, which corresponds to one state of the automaton in Fig. 3(b), the control loop (see section III-B) is activated to adapt the parallelism or suspend the parallelism adaptation. A *profile length* is a fixed period (marked by logic time: the number of commits) for information gathering, such as commits, aborts and time. A *parallelism profile interval* is composed of a continuous sequence of profile lengths within which the parallelism is adjusted and the CR range is computed. The *non-action interval* consists of one or a continuous sequence of profile lengths, within which the parallelism adaptation is suspended. The duration of parallelism profile interval and non-action interval are not fixed values as shown in Fig. 2. The above procedure continues until the program terminates.

B. Feedback Control Loop of the Simple Model



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



(b) The structure for the autonomic manager of Fig. 3(a) in automaton shape.

Fig. 3. **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 transfer.**

Fig. 3(a) gives the structure of the complete platform that forms 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. 3(b). The automaton consists of four states, and the program can only reside on one state at each decision point.

1) Control Objective: Under control theory terminology, the control objective of the feedback control loop is to maximize the throughput and diminish the global execution time.

2) Inputs and Outputs: As shown in Fig. 3(a), the inputs are commits, aborts, physical time and current active thread number. The outputs are the optimum parallelism, the parallelism profile flag.

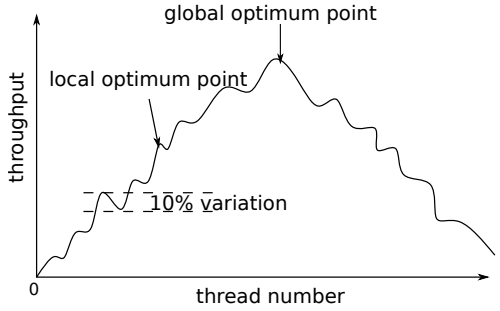


Fig. 4. **Throughput fluctuation. The throughput may continuously rise and descend before reaching its maximum point.**

3) Decision Functions: Three decision functions cooperate to make decisions: a *parallelism decision function* (adjusts parallelism), a *profile decision function* (enable the parallelism profile actions) and a *CR range decision function* (dictates the phase variation). We describe design of the automaton as it elucidates the relation among the decision functions, as well as how the parallelism decision function and profile decision function are designed. The CR range decision function is presently lastly.

The automaton commences at the state *increase tn* to increase the thread number, since the parallelism is set to be the minimum at the starting point. In each parallelism profile interval, the parallelism can either continuously increase or decrease. The direction of parallelism regulation (increase or decrease) is determined by the profile decision function. If the current throughput is greater than the previous throughput, one thread is awaken or suspended and the current throughput is recorded as the maximum value. The state transfers to *stop profile* when the current throughput is less than the maximum throughput. At the final decision point of a parallelism profile interval (the state *stop profile*), the parallelism is set to be the value which yields the maximum throughput. A new CR range may be determined at the end of a parallelism profile interval, as we will detail later in this section. The automaton then enters from the state *stop profile* to the state *no tn control* which corresponds to the non-action interval in Fig. 2. At each decision point of a non-action interval, the profile decision function checks if it should start parallelism adaptation. More specifically, if the CR falls into the CR range, the program stays in the *no tn control* state. Otherwise a boolean value is set indicating the direction of the parallelism regulation. The automaton jumps into *increase tn* if CR is higher than the upper CR threshold or *decrease tn* if CR is lower than the lower CR threshold. It is worth noting that, in case the value of the upper threshold is 100%, and the program CR is also 100% (it happens when only read operations or no conflicts across transactions), higher parallelism is assigned to the program.

It is worth noting that the throughput often fluctuates before reaching the optimum value (shown in Fig. 4). 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 (10% is an empirical value which can be tuned).

The *parallelism decision function* resides in the 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 *CR range decision function*. Therefore at the end of a parallelism interval, a new CR range is prescribed. The function is activated at the state *stop profile*. 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 one.

C. Probabilistic Model

The approach to manipulate one thread number at each decision point engages long profiling time. This section presents a probabilistic model which predicts favourable parallelism after one profile length based on the CR and current active thread number. The profiling procedure is similar to the simple model as shown in Fig. 2.

The probabilistic model shares the same inputs, outputs and control objective as the simple model. It also incorporates three decision functions. As the *CR range decision function* is equivalent to that in Section III-B3, we only describe the *parallelism decision function* and the *profile decision function* in this section.

1) Decision Functions: We firstly present the *parallelism decision function*. It is based on two assumptions:

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

The detail of the derivation of the decision function is presented in [10], we only provide the final equation here. The optimum thread number is calculated by (1):

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

Where n_{opt} represents the optimum parallelism, n represents the number of running threads and CR is the commit ratio.

In this paragraph, we describe the automaton to elucidate the relation of the three decision functions as well as the design of the profile decision function. As illustrated in Fig. 5, the automaton commences (with the maximum parallelism) from the *predict tn* state which yields an estimation of optimum parallelism degree. The predicted parallelism is applied for one subsequent profile length and the automaton unconditionally enters the *verify* state to verify the correctness of the predicted parallelism. The new parallelism is only applied subsequently when the current throughput is larger than the throughput recorded before the new parallelism is applied. This leads to the state transfer to the *CR range* state where a new CR range is prescribed. The *stop profile* state only disables the parallelism

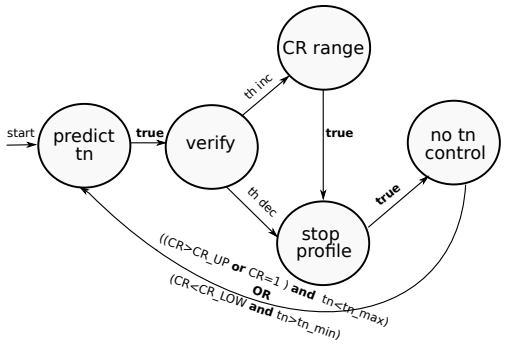


Fig. 5. **The controller structure of the probabilistic model described as an automaton.**

profile action when the parallelism does not alter. Otherwise it recovers the previous value (CR range remains unchanged in this case). Contrary to the simple model, the probabilistic model requires an individual state to obtain the CR range.

IV. Implementation

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

We implement a monitor to collect the profile information, adjust the parallelism and the race condition. The monitor is a cross-thread lock which consists of concurrent-access variables by threads. The main variables of the monitor are commits, aborts, two FIFO queues recording the suspended and active threads, the current active thread number, the optimum thread number and the throughput. There are three entry points of the monitor. The first entry point is upon threads initialization, where some initial values (*e.g.*, thread id) are set for the threads and some threads are suspended. The second entry point is upon transaction committing, where commits are accumulated and where the control functions take actions. The third entry point is upon a thread exiting, where one suspended thread is awaken when one thread exits.

We use logic time (number of commits) to mark the profiling length rather than the physical time. As the size of transaction varies in various applications leading to the significant variation of execution time. The choice of the profile length mainly depends on the total amount of transactions of an application. The applications with the same magnitude of transactions share the same profile length. For instance, **genome** and **vacation** (two benchmarks from **STAMP**) share the same profile length as the total number of the transactions in the two applications are on the same magnitude (10^6).

Time overhead is added to each transaction when calling and releasing a monitor. The overhead is insignificant for the transactions with medium or long length, however it gives non-trivial influence to the short-length transactions. This 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.

V. Performance Evaluation

In this section, we present the results from 6 different **STAMP** [11] benchmarks and two applications from **EigenBench** [12]. **EigenBench** and **STAMP** are widely used for performance evaluation on TM systems. The data sets cover a wide range from short-length to long-length transaction, short to long program execution time, from low to high program contention. Table I presents the qualitative summary of each application's runtime transactional characteristics: TX length (the number of instructions per transaction), execution time, and contention (the global contention). The classification is based on the application with its static optimal parallelism. A transaction with execution time between 10 *us* and 1000 *us* is classified as medium-length. The contention between 30% and 60% is classified as medium. The execution time between 10 seconds and 30 seconds is classified as medium.

TABLE I. **Qualitative summary of each application's runtime transactional characteristics. The classification is based on the application with its optimal parallelism applied.**

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

A. Platform

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

B. Benchmark Settings

We show two different data sets of **EigenBench**. One with stable behaviour and one with diverse phases. As **EigenBench** does not shows phase variation, we modify its source code to enable diverse phases. **EigenBench** include 3 different arrays which provide the shared transactional accesses (Array1), private transactional accesses (Array2) and non-transactional accesses (Array3). We vary the size of Array1 at runtime making the conflict rate vary at runtime. More specifically, within the first 40% amount of the transactions, the size of Array1 keeps the value given by the input file. From 40% to 70%, the array size is shrunk to be 16% of the original value and afterwards the size is set to be 33% of the given value. The main inputs of both data sets are given in Fig. 6.

We have evaluated 6 different applications from **STAMP**, namely **intruder**, **ssca2**, **genome**, **vacation**, **yada** and

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) stable

loops	33333
A1	145530
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

(b) online variation

Fig. 6. **Two data sets of EigenBench inputs for 24 threads**

labyrinth. Two applications namely **bayes** and **kmeans** from **STAMP** are not taken into account in the paper. As **bayes** exhibits non-determinism [13]: the ordering of commits among threads at the beginning of an execution can dramatically affect the execution time. The aforementioned applications have represented a wide range of characteristics of TM applications, therefore we do not present the results from **kmeans** due to the page limit. The inputs of the six selected applications are detailed in Fig. 7.

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

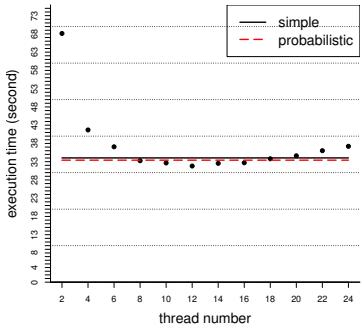
Fig. 7. **The inputs of STAMP**

C. Results

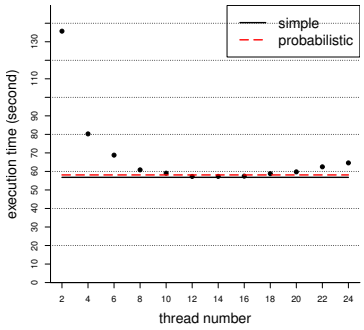
We firstly present the results of the execution time comparison of the two autonomic parallelism adaptation approaches with static parallelism. The maximum parallelism is 24 which is the number of the available cores. The minimum parallelism is restricted to be 2, as we are only concerned with parallel applications. All the applications are executed 10 times and results are the average execution time.

Fig. 8 and Fig. 9 illustrate the execution time comparison with different static parallelism and adaptive parallelism of **EigenBench** and **STAMP**. The dots represent the execution time with static parallelism. The solid black line represents execution time with the simple model and the dashed red line gives the execution time with the probabilistic model.

According to Fig. 8 and Fig. 9, our adaptive models outperform the performance of the majority of the static parallelism. The probabilistic model shows better performance on applications: **genome**, **vacation**, **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**. Table II and Table III detail the performance comparison. The results of both models are compared against the static parallelism which presents best, average and worst performance indicating that our models can outperform the performance of static parallelism if an unknown application is given. The digits in the



(a) the data set with stable online behaviour



(b) the data set with online variation

Fig. 8. **Time comparison of EigenBench on static and adaptive parallelisms. The dots represent the execution time with static parallelism**

brackets are the static parallelism which gives the best and the worst performance respectively. The symbol plus (+) means performance gain against the compared value.

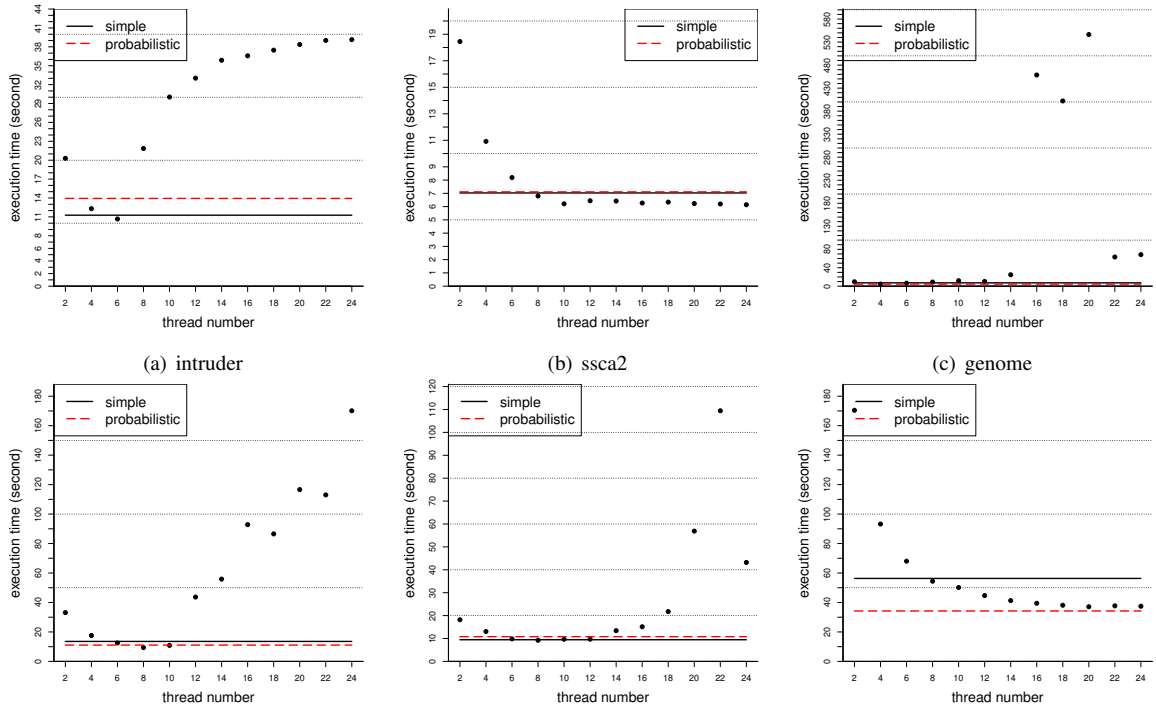
TABLE II. **Performance comparison of simple model against static parallelism on applications. The higher value, the better performance.**

benchmarks	best case	average	worse case
EigenBench (stable)	-7% (12)	+10%	+50% (2)
EigenBench (online variation)	+1% (12)	+17%	+58% (2)
genome	-57% (4)	+95%	+99% (20)
vacation	-45% (8)	+79%	+92% (24)
labyrinth	-52% (24)	+5%	+67% (2)
yada	-3% (8)	+66%	+91% (22)
ssca2	-14% (24)	+11%	+62% (2)
intruder	-6% (6)	+62%	+71% (24)

TABLE III. **Performance comparison of probabilistic model against static parallelism on applications. The higher value, the better performance**

benchmarks	best case:	average	worse case
EigenBench (stable)	-5% (12)	+11%	+51% (2)
EigenBench (online variation)	-1% (12)	+18%	+57% (2)
genome	+3% (4)	+97%	+99% (20)
vacation	-18% (8)	+83%	+93% (24)
labyrinth	+8% (24)	+42%	+80% (2)
yada	-17% (8)	+61%	+90% (22)
ssca2	-16% (24)	+10%	+61% (2)
intruder	-31% (6)	+53%	+64% (24)

Fig. 10 elucidates the runtime parallelism variation with simple and probabilistic model of **genome**. **genome** experiences three phases at runtime. The first phase is short (two or three profile lengths) which contains both read and write operations. During the second phase, the transactions only



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

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

Fig. 9. **Time comparison for STAMP on static and adaptive parallelism. The dots represent the execution time with static parallelisms.**

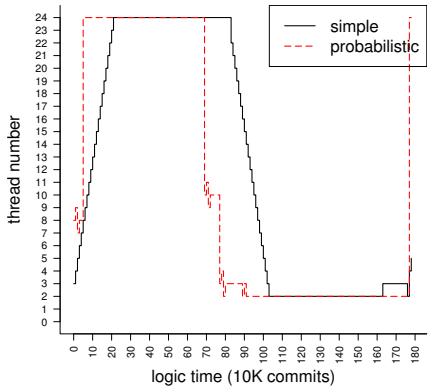


Fig. 10. **Runtime parallelism variation by the two models of genome.**

include read operations resulting in 100% of CR, hence the maximum parallelism is applied. The third phase generates high contention, hence low parallelism is given. As shown in Fig. 10, 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. As less time spent to reach the optimum parallelism, the performance of probabilistic model outperforms that of the simple 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. Due to the page limit, we only present the online throughput change of one application **genome**. Fig. 11 elucidates its online throughput change with static and adaptive parallelism.

VI. Discussion

The overhead of our approaches mainly originate in three aspects. (1) Thread migration. This can introduce a large overhead especially when the parallelism is adapted at runtime. (2) The choice of the thread number to manipulate at each decision point in the simple model. We simply choose to manipulate one thread number each point which delays the procedure to reach the optimum parallelism. (3) The choice of throughput variation rate in the simple model. We keep profiling even if the current throughput is slightly lower than the recorded maximum value in order to avoid the parallelism profiling to be terminated at a regional maximum point.

Performance penalty can occur when parallelism varies, yet is trivial on shared memory. In addition, to reduce the penalty, a thread is only suspended when its current transaction commits. The CR range decision function shows its limitation on phase detection for **vacation**. As its CR tend to fluctuate frequently over the CR range, yet the program remains in the same phase. Therefore both model perform insufficiently **vacation**.

The simple model only manipulates one thread number at each decision point thus long time spent in searching the optimum parallelism. However this eludes the possibility of skipping the optimum parallelism. The application starts with

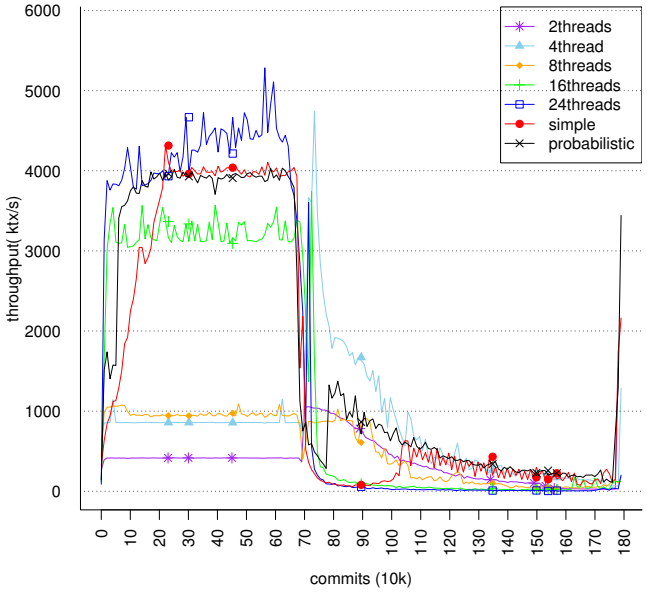


Fig. 11. **Runtime throughput variation of genome. The red line with dots is the simple model and the black line with crosses is the probabilistic model.**

two threads activated rather than the maximum value to avoid excessive contention which may prevent the program from progressing. However this setting brings high parallelism profiling time to the applications which require a high parallelism, this is especially true for **labyrinth** that requires maximum parallelism to achieve its maximum throughput. Such an overhead is difficult to be compensated by the performance improvement brought by the optimum parallelism. Therefore, the simple model gives heavy overhead against the best case to the application with long-length transactions. **genome** requires the maximum parallelism at the second phase but minimal parallelism at the third phase leading to a high overhead originating in parallelism descending. Such overhead is especially large when the application generates high contention.

The probabilistic model predicts the parallelism in one step which diminishes the profiling time and often gives better performance than the simple model. However it has the potential risk of overreacting to the phase variation as **intruder**. Lastly, the probabilistic model relies on two assumptions which are based on the ideal situations, thus some errors are imposed inevitably in reality, meaning that the predicted parallelism may be slightly different as the optimum one. Thereby when phase changes are trivial, the simple model can outperform the probabilistic model (*e.g.*, **yada**).

VII. Conclusion and Future Work

In this paper, we investigate two autonomic parallelism adaptation approaches on a STM system. We examine the performance of different static parallelism and conclude that runtime parallelism adaptation is crucial to performance of TM applications. We introduce feedback control loops to manipulate the parallelism. Followed the description of our models, we compare their performance with static parallelism. We then analyse the implementation overhead and discussed the advantages as well as limitation of our work.

Apart from inappropriate parallelism, thread migration impacts on system performance and cause performance degradation. We plan to investigate the issue and design additional control loops that cooperate with the current loops to control thread affinity and further enhance system performance.

Acknowledgment

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

References

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

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

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

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

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

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

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

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

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

[10] N. Zhou, G. Delaval, B. Robu, É. Rutten, and J.-F. Méhaut, "Autonomic Parallelism Adaptation on Software Transactional Memory," Research Report RR-8887, Univ. Grenoble Alpes ; INRIA Grenoble, Mar. 2016.

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

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

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