Sapienza University of Rome

Ph.D. program in Computer Engineering

XXVI Cycle

Autonomic Concurrency Regulation in Software Transactional Memories

Diego Rughetti

2014/3

Sapienza University of Rome

Ph.D. program in Computer Engineering

XXVI Cycle

Diego Rughetti

Autonomic Concurrency Regulation in Software Transactional Memories

Thesis Committee	**Reviewers**
Prof. Bruno Ciciani (Advisor)	Prof. Vincent Gramoli
Prof. Giorgio Grisetti	Prof. Jean-François Méhaut

2014/3

Author's address:
Diego Rughetti
Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Sapienza Università di Roma
Via Ariosto 25, 00185 Roma, Italy
e-mail: `rughetti@dis.uniroma1.it`

www: `http://www.dis.uniroma1.it/~rughetti/`

To my grandparents

Contents

1 Introduction **7**

2 Software Transactional Memories **13**

2.1 Software Transactional Memories brief overview 13

2.2 The problem of concurrency in STM 15

3 Performance Optimization in STM, State of Art **27**

3.1 Approaches for performance optimization in STM 31

4 Application Model and Workload Features **45**

4.1 Model of the STM application 45

4.2 STM application performance model 46

4.3 Input parameters analysis 47

4.4 Input parameters - experimental data 50

5 Machine Learning Based Approach **55**

5.1 System Architecture 57

5.1.1 Controlling the Concurrency Level 59

5.2 Implementation 63

5.3 Experimental Evaluation 65
5.3.1 Evaluation Methodology 66
5.3.2 Off-line Training 67
5.3.3 Results 70
5.4 Sampling Overhead Optimization 79
5.5 Dynamic feature selection 82
5.5.1 Rationale 82
5.5.2 Pragmatic Relevance: Run-time Sampling Costs 85
5.5.3 Shrinking vs Enlarging the Feature Set 92
5.5.4 The Actual Dynamic Feature Selection Architecture . . . 95
5.6 Experimental Evaluation 98

6 The Analytical Model Based Approach 107
6.1 The Parametric Performance Model 108
6.1.1 Basic Assumptions 109
6.1.2 Model Construction 109
6.1.3 Instantiating ρ, ω and ϕ 111
6.1.4 Model Validation with Respect to a Real System 122
6.2 Concurrency Self-Regulating STM 127
6.2.1 The Architecture 127
6.2.2 Evaluation Study 130

7 The Hybrid Approach 135
7.1 Performance Model Aim 137
7.2 The Actual AML Model 139
7.2.1 Correcting Factors 145
7.3 Experimental Evaluation 147

7.3.1 The AML Based Concurrency Regulation Architecture . . 147
7.3.2 Experimental Data 149

8 STMs Performance Comparison **167**

8.1 Reference Architectures and Configurations 168
8.1.1 SwissTM 168
8.1.2 TinySTM 169
8.1.3 TinySTM Adaptive Configurations 170
8.2 Experimental Data 172

9 Conclusions **185**

A Neural Networks **191**

B The STAMP Benchmark Suite **195**

List of Figures

2.1 Total application execution time varying the concurrency level . 23

4.1 Total mean execution time vs. mean execution time 51
4.2 Total mean execution time vs. mean non-transactional time . . . 51
4.3 Total mean execution time vs. mean read write set size 52
4.4 Total mean execution time vs. mean read set size 52
4.5 Total mean execution time vs. write-write index 52
4.6 Total mean execution time vs. read-write index 53

5.1 System Architecture. 58
5.2 Stalled cycles back-end for the intruder benchmark 61
5.3 Stalled cycles back-end for the vacation benchmark 61
5.4 Example training configurations and test configurations for the case of three workload configuration parameters. 66
5.5 Average wasted transaction execution time: training set vs. predicted. 68
5.6 Average wasted transaction execution time: measured vs. predicted. 68

5.7 Application execution time - intruder 71
5.8 Application execution time - genome 73
5.9 Application execution time - kmeans 74
5.10 Application execution time - labyrinth 75
5.11 Application execution time - ssca2 75
5.12 Application execution time - vacation 76
5.13 Application execution - yada 77
5.14 Application execution - bayes 78
5.15 Application execution time with intruder up to 32 threads. . . . 80
5.16 Relative and absolute instrumentation costs - intruder 89
5.17 Relative and absolute instrumentation costs - bayes 90
5.18 Relative and absolute instrumentation costs - ssca2 90
5.19 Relative and absolute instrumentation costs - vacation 90
5.20 Relative and absolute instrumentation costs - kmeans 91
5.21 Relative and absolute instrumentation costs - labyrinth 91
5.22 Relative and absolute instrumentation costs - yada 92
5.23 Relative and absolute instrumentation costs - genome 93
5.24 Relation between the final achieved performance and WRMS for vacation 95
5.25 Relation between the final achieved performance and WRMS for intruder 96
5.26 Extended SAC-STM architecture 98
5.27 Results for intruder 100
5.28 Results for ssca2 101
5.29 Results for vacation 102

5.30 Parameters and throughput variation over time for the modified vacation benchmark 103
5.31 Results for modified vacation 104

6.1 Variation of ρ with respect to the write-write affinity 114
6.2 Variation of ρ with respect to the write-set size 114
6.3 Simulated vs predicted abort probability while varying ww_a and ws_s 116
6.4 Variation of ρ with respect to the read-write affinity 117
6.5 Variation of ρ with respect to the read-set size 118
6.6 Simulated vs predicted abort probability vs rw_a, rs_s, ww_a, ws_s . 119
6.7 Variation of ω vs the number of concurrent threads 120
6.8 Simulated vs predicted abort probability vs k, rw_a, rs_s, ww_a, ws_s 120
6.9 Variation of ϕ with respect to θ 121
6.10 Simulated vs predicted abort probability while varying all the workload profile parameters 121
6.11 Model and neural-network prediction accuracy 126
6.12 CSR-STM architecture 127
6.13 Execution time for vacation with CSR-STM and TinySTM . . . 131
6.14 Execution time for kmeans with CSR-STM and TinySTM 132
6.15 Execution time for yada with CSR-STM and TinySTM 132

7.1 System architecture 146
7.2 Predictions accuracy - yada benchmark 153
7.3 Predictions accuracy - kmeans benchmark 153
7.4 Prediction error comparison - vacation 155
7.5 Prediction error comparison - intruder 156

7.6 Prediction error comparison - yada 156
7.7 Prediction error comparison - kmeans 157
7.8 Throughput - vacation 159
7.9 Throughput - intruder 159
7.10 Throughput - yada 160
7.11 Throughput - kmeans 160
7.12 Energy consumption per committed transaction - intruder 162
7.13 Energy consumption per committed transaction - kmeans 162
7.14 Energy consumption per committed transaction - yada 163
7.15 Energy consumption per committed transaction - vacation 163
7.16 Iso-energy speedup curves 164

8.1 Application execution time - intruder benchmark 174
8.2 Application execution time - genome benchmark 175
8.3 Application execution time - kmeans benchmark 176
8.4 Application execution time - ssca2 benchmark 178
8.5 Application execution time - vacation benchmark 179
8.6 Application execution time - labyrinth benchmark 180
8.7 Application execution time - yada benchmark 181
8.8 Application execution time - bayes benchmark 183

A.1 Example of a three layer fully connected neural network 193

List of Tables

5.1 Input features correlation for the applications with the STAMP benchmark suite 86
5.2 Input features variance for the applications with the STAMP benchmark suite 87
5.3 Parameters configuration for the performance tests 99
5.4 Minimum achieved percentage of ideal speedup 105

6.1 Abort probability prediction error (and its variance) 125

7.1 Average error and variance comparison 151

B.1 Qualitative summary of STAMP application's runtime transactional characteristics 200

Acknowledgements

My first special thanks goes to my colleagues and friends Pierangelo Di Sanzo and Alessandro Pellegrini.

Thanks Piero because without your help probably I'd still be looking for a good research topic for my Phd, because without your being stubborn many of the our papers would not be accepted and because you always find the right moment to say "sta chiudendo!".

Thanks Alessandro because without your help probably I'd still be trying to remove some "segmentation fault" error from my code, because you provided to me the right psychological support during my first important international conference in Washington D.C. and because you have been my official cigarettes provider.

A special thanks to my advisor and leader of our research group Prof. Bruno Ciciani, who is more than just a great scientist, but a master of life. He is like a father for all the guys of our research group and he transmitted to us the enthusiasm for the teaching. He taught me not only the approach to scientific research but to be a honest person and to never compromise myself for any reason.

A special thanks to Prof. Francesco Quaglia, that I consider like a second advisor. I met him some years ago, when I was a student of bachelor degree, and I immediately perceived the reliability and the competence that distinguish him. During the last years he gave to me the right suggestions to solve the problems that manifested during my doctorate and he taught to me that the quality of the work is the only target that must be addressed. He is a special person, despite some serious personal problems he has always found the time to work with us. Thank you very much Francesco and always "Forza Roma"!

Together with Bruno and Francesco, a special thanks goes to Prof. Paolo Romano. He was my advisor for the master degree thesis and he gave me the opportunity to have a wonderful research experience in Lisbon. He has been a reference point for his way to approach and solve the problems. Paolo please say thank you to Anna for the wonderful stuffed squid she coked for us when I was in Lisbon!

I am also grateful to Prof. Vincent Gramoli and Prof. Jean-François Méhaut for having accepted to serve as external referees of this dissertation, and to Prof. Giorgio Grisetti for having served as second member of my thesis committee.

A special thanks to the young researchers who have been close to me in any situation (Roberto Vitali, Roberto Palmieri and Sebastiano Peluso): their jokes and laughters greatly alleviated the stress experienced during the whole PhD program.

A loving thanks to my family (Rosa, Renzo and Simone), the only reason that made it possible to achieve this result. They dedicated their entire life to me and my brother. I wish I knew some way of returning even a fraction of what they have given me.

A final thanks to all the people that gave and thought to me something important, for my work and for my life: Valeria Cruciani, Andrea "Pasticca"

Bassignani, Emiliano “Pera” Di Giambattista, Fabio “Francis” Francescangeli, Alessia Iacuitto, Sara Cherubini, Daniele “Dang” D’angeli, Lucio D’Angeli, Riccardo “Ninetto” Colantoni, Veronica Cavalli, Luca Pascasi, Federico “Generale” Boni, Andrea “Zione” Capata, Cristina Canu, Tatiana Lopez, Leonardo “Mister” Bevilacqua, Sabrina D’auria, Marco “Il Mentore” Martini and Francesca Casciola.

Rome, Italy
March 31th, 2014

Diego

Abstract

Software Transactional Memory (STM) has emerged as a powerful programming paradigm for concurrent applications. It allows encapsulating the access to data shared across concurrent threads within transactions, thus avoiding the need for synchronization mechanisms to be explicitly coded by the programmer. On the other hand, synchronization transparency must not come the expense of performance. Hence, STM-based systems must be enriched with mechanisms providing optimized run-time efficiency. Among the issues to be tackled, a core one is related to determining the optimal level of concurrency (number of threads) to be employed for running the application on top of the STM layer. For too low levels of concurrency, parallelism can be hampered. On the other hand, over-dimensioning the concurrency level may give rise to thrashing phenomena caused by excessive data contention and consequent transaction aborts.

In this thesis we propose a set of techniques in order to build "application specific" performance models allowing to dynamically tune the level of concurrency to the best suited value depending of the specific execution phase of the application. We will present three different approaches: a) one based on a pure Machine Learning (ML) model that doesn't require a detailed knowledge of the

application internals to predict the optimal concurrency level, b) one based on a parametric analytical performance model customized for a specific application/-platform through regression analysis that, respect to the previous one, requires a lighter training phase and c) one based on a combination of analytical and Machine Learning techniques, that allows to combine the strengths of the previous two approaches, that is it has the advantage of reducing the training time of pure machine learning methods avoiding the approximation errors typically affecting pure analytical approaches. Hence it allows very fast construction of highly reliable performance models, which can be promptly and effectively exploited for optimizing actual application runs.

We also present real implementations of concurrency regulation architectures, based on our performance predictions approaches, which have been integrated within the open source TinySTM package, together with experimental data related to runs of application profiles taken from the STAMP benchmark suite demonstrating the effectiveness of our proposals. The experimental data confirm how our self-adjusting concurrency schemes constantly provides optimal performance, thus avoiding performance loss phases caused by non-suited selection of the amount of concurrent threads and associated with the above depicted phenomena.

Moreover we present a mechanism that allows to dynamically shrinks or enlarges the set of input features to be exploited by the performance predictors. This allows for tuning the concurrency level while also minimizing the overhead for input-features sampling, given that the cardinality of the input-feature set is always tuned to the minimum value that still guarantees reliability of workload characterization. We also present a fully fledged implementation of this solution again within the TinySTM open source framework, and we provide the results

of an experimental study relying on the STAMP benchmark suite, which show significant reduction of the application execution time with respect to proposals based on static feature selection.

Publications

Most of the material provided by this dissertation has been presented in (or has contributed to the production of) the following technical articles I have coauthored:

1 Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani, Francesco Quaglia Analytical/ML Mixed Approach for Concurrency Regulation in Software Transactional Memory. *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, (CCGrid). IEEE/ACM Chicago, IL, USA May 26-29, 2014

2 Diego Rughetti, Perangelo Di Sanzo, Bruno Ciciani, Francesco Quaglia. Machine Learning-based Self-adjusting Concurrency in Software Transactional Memory Systems. *Proceedings of the 20th IEEE International Symposium On Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, (MASCOTS). pages 278-285, IEEE Computer Society, August 2012, Arlington, VA, USA

3 Pierangelo Di Sanzo, Francesco Del Re, Diego Rughetti, Bruno Ciciani and Francesco Quaglia Regulating Concurrency in Software Transactional Memory: An Effective Model-based Approach. *Proceedings of the 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, (SASO). IEEE Computer Society, September 2013, Philadelphia, USA

4 Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani, Francesco Quaglia Dynamic Feature Selection for Machine-Learning Based Concurrency Regulation in STM. *Proceedings of the 22st Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP Euromicro, University of Turin, Turin, Italy, February 2014

5 Diego Rughetti, Pierangelo Di Sanzo and Alessandro Pellegrini Adaptive Transactional Memories: Performance and Energy Consumption Tradeoffs. *Proceedings of the 3rd IEEE Symposium on Network Cloud Computing and Applications*, (NCCA). IEEE Computer Society, February 2014, Rome, Italy

The techniques presented in this dissertation have been used also in context different from Software Transactional Memory (e.g. Hardware Transactional Memory and Data Grid platforms deployed in Cloud Computing environments) bringing to the production of the following technical articles I have coauthored:

1 Diego Rughetti, Paolo Romano, Francesco Quaglia, Bruno Ciciani. Autonomic Tuning of the Parallelism Degree in Hardware Transactional Memory. SUBMITTED to the *20th International European Conference on Parallel Processing*, (Euro-Par). Springer, August 2014, Porto, Portugal

2 Pierangelo Di Sanzo, Francesco Maria Molfese, Diego Rughetti, Bruno Ciciani Providing Transaction Class-Based QoS in in-Memory Data Grids Via Machine Learning. *Proceedings of the 3rd IEEE Symposium on Network Cloud Computing and Applications*, (NCCA). IEEE Computer Society, February 2014, Rome, Italy

3 Perangelo Di Sanzo, Diego Rughetti, Bruno Ciciani, Francesco Quaglia. Auto-tuning of Cloud-based In-memory Transactional Data Grids via Ma-

chine Learning. *Proceedings of the 2nd IEEE Symposium on Network Cloud Computing and Applications*, (NCCA). IEEE Computer Society, December 2012, London, UK

4 Perangelo Di Sanzo, Francesco Antonacci, Bruno Ciciani, Roberto Palmieri, Alessandro Pellegrini, Sebastiano Peluso, Francesco Quaglia, Diego Rughetti, Roberto Vitali. A Framework for High Performance Simulation of Transactional Data Grid Platforms. *Proceedings of the 6th ICST Conference of Simulation Tools and Techniques*, (SIMUTools). ICST, March 2013, Nice, France

5 Paolo Romano, Diego Rughetti, Bruno Ciciani, Francesco Quaglia. APART: Low Cost Active Replication for Multi-tier Data Acquisition Systems. *Proceedings of the 7th IEEE International Symposium on Network Computing and Applications*, (NCA). IEEE Computer Society, July 2008, Cambridge, MA, USA, Winner of the Best Paper Award

Chapter 1

Introduction

Software Transactional Memory (STM) [1] is an attractive programming paradigm for parallel/concurrent applications. Particularly, by relying on the notion of atomic transaction, STM stands as a friendly alternative to traditional lock-based synchronization. More in detail, code blocks accessing shared-data can be marked as transactions, thus demanding coherency of data access/manipulation to the STM layer, rather than to any handcrafted synchronization scheme provided by the programmer. The relevance of the STM paradigm has significantly grown given that multi-core systems have become mainstream platforms, so that even entry-level desktop and laptop machines are nowadays equipped with multiple processors and/or CPU-cores. Also, transaction is the representative technology for several in-memory Cloud-suited data-platforms (such as Red Hat's Infinispan, VMware vFabric GemFire [2], Oracle Coherence [3] and Apache Cassandra [4]), where the encapsulation of application code within transactions allows concurrent manipulation of in-memory kept application data according to specific isolation levels, which is done transparently to the programmer.

Even though the STM potential for simplifying the software development process is extremely high, another aspect that is central for the success, and the further diffusion of the STM paradigm relates to the actual level of performance it can deliver. As for this aspect, one core issue to cope with in STM is related to exploiting parallelism while also avoiding thrashing phenomena due to excessive transaction rollbacks, caused by excessive contention on logical resources, namely concurrently accessed data portions. We note that this aspect has reflections also on the side of resource provisioning in the Cloud, and associated costs, since thrashing leads to suboptimal usage of resources (including energy) by, e.g., PaaS providers offering STM based platforms to customers. One such platform has been recently presented by the Cloud-TM project [5], and is aimed at simplifying and optimizing the process of deploying data centric applications in the Cloud.

In order to deal with the run-time efficiency issue in STM, literature approaches can be framed within different sets of orthogonal solutions. On one side can we find optimized schemes for transaction conflict detection and management [6, 7, 8, 9, 10, 11]. These include proposals aimed at dynamically determining which threads need to execute specific transactions, so to allow transactions to be expected to access the same data to run along a same thread in order to sequentialize and spare them from incurring the risk of being aborted with high probability. Other proposals rely instead on pro-active transaction scheduling [12, 13] where the reduction of performance degradation due to transaction aborts is achieved by avoiding to schedule (hence delaying) the execution of transactions whose associated conflict probability is estimated to be high. All the above schemes are not meant to optimize the concurrency level (that is the number of threads) to be used for running the application, thus they generally operate on top of configurations where the number of threads is predetermined at application startup.

On the other side we find solutions aimed at supporting performance optimization via the determination of the best suited level of concurrency to be exploited for running the application on top of the STM layer (see, e.g., [14, 15, 16]). These solutions are clearly orthogonal to the aforementioned ones, being potentially usable in combination with them. We can further distinguish these approaches depending on whether they can cope with dynamic or static application execution profiles, and on the type of methodology that is used to determine (predict) the well suited level of concurrency for a specific (phase of the execution of the) application. Approaches coping with static workload profiles are not able to predict the optimal level of concurrency for applications where classical parameters expressing proper dynamics of the applications (such as the average number of data-objects touched by a transactional code bock) can vary over time. For those scenarios approaches coping with dynamic workload profiles usually allows to obtain better prediction performance.

The prediction approaches that have been proposed in literature either rely on analytical methods, or on black-box Machine Learning (ML) methodologies. The first ones have the advantage of generally requiring a lightweight application profiling for gathering data to be filled to the prediction model, but provide (slightly) less accurate predictions and in some cases require stringent assumptions to be met by the real STM system in order for its dynamics to be reliably captured by the analytical formulas (see, e.g., [17]). On the contrary, ML methods usually require much expensive profiling in order to build the knowledge base that would suffice to instantiate the performance prediction model, which may make the actuation of the optimized concurrency configuration untimely. On the other hand, they typically allow high precise estimation of the real performance trends of the STM system as a function of differentiated parameters (see, e.g., [18]).

In this thesis we cope with the issue of determining the optimal level of concurrency by presenting a suite of techniques for dynamic concurrency regulation based on Machine learning, on analytical modelling and on an hybrid technique that mixes the previous two approaches allowing to chase the best of the two methodologies by tackling the shortcomings intrinsic in each of them. The approaches we provide are able to cope with cases where the actual execution profile of the application, namely the workload features, can change over time, such as when the (average) size of the data-set accessed by the transactional code in read or write mode changes over time (e.g. according to a phase-behavior). This is not always allowed by pure analytical approaches [14, 16]. Furthermore, as we will show later in this thesis, our hybrid approach represents a methodology for very fast construction of a highly reliable performance model allowing the determination of the optimal level of concurrency for the specific STM-based application. This is relevant in generic contexts also including the Cloud, where the need for deploying new applications (or applications with reshuffling in their execution profile), while also promptly determining the system configurations allowing optimized resource usage, is very common.

During the development of our concurrency regulation techniques we verified that one drawback of the ones that use Machine Learning is related to the need for constantly monitoring the set of selected input features to be exploited by the machine learner. This may give rise to non-minimal overhead, especially when considering that STM applications may exhibit fine-grain transactions, natively requiring a (very) reduced amount of CPU cycles for finalizing their task. To cope with this issue, we developed a solution where the set of input features exploited by the machine learning based performance model is dynamically shrunk.

In other terms, the complexity of both the workload characterization model and the associated performance model is (dynamically) reduced to the minimum that still guarantees reliable performance prediction. This leads to reducing the amount of feature samples to be taken for performance prediction along any wall-clock-time window, hence reducing the actual overhead for performance prediction.

Together with the aforementioned approaches, we also present three real implementations of concurrency regulation architectures, integrated with the TinySTM open source package [19], which exploit the developed models to dynamically tune the number of threads to be used for running the application. Further, we report experimental results, achieved by running the applications belonging to the STAMP benchmark suite [20] on top of a 16-core HP ProLiant machine, which show the effectiveness of the proposed approaches.

The reminder of the thesis is organized as follows. In Chapter 2 a description of software transactional memories and of the concurrency regulation problem in STM based applications is provided. The state of art about performance optimization in STM is discussed in Chapter 3. In Chapter 4 the models for the target STM applications and for the transactional workload are presented. In Chapters 5, 6, 7 three innovative approaches for autonomic concurrency regulation based respectively on machine learning, analytical modelling and a mix of the previous two are presented and discussed. We conclude the thesis presenting in Chapter 8 a detailed comparison between the performance reachable using different standard and adaptive STM implementations.

To help the reader to better understand the insights of the approach presented in Chapter 5 we provide in Appendix A a recall of the used machine learning technique. Moreover a brief overview of the STAMP benchmark suite, the most commonly used testbed for STM platforms, is provided in Appendix B.

Chapter 2

Software Transactional Memories

2.1 Software Transactional Memories brief overview

Software transactional memories (STMs) [1] are a high attractive programming paradigm for parallel applications. The first proposals about Transactional Memories (TMs) [8] were dated back to 90s but research on this topic gained momentum since 2004, when the multi-core processors became available for commercial market. Proliferation of multi-core architectures allowed parallel programming to exit from the niche of high-performance and scientific computing and turned it into a mainstream concern for software industry. One of the main challenge of parallel programming is the synchronization of concurrent accesses to shared memory by multiple, concurrent threads. A traditional technique is the one based on locks, but it has well-known pitfalls: sophisticated fine-grained locking can bring to the risk of deadlocks and data races while more simple coarse-grained locking can bring to scalability limitations. Moreover scalable libraries that use fine-grained lock cannot be easily mixed in a way that preserve scalability and avoid data races and deadlock [21]. STMs inherit the transaction based approach typical of Database systems (DBS), bringing it into the world of parallel programming. They free the programmers from the responsibility of design and verify complex fine-grained lock synchronization schemes. By avoiding deadlocks and automatically allowing fine-grained concurrency, transactional language constructs enable the programmer to compose scalable applications safely out of thread-safe libraries. As in SQL programming, with STM the programmers just have to mark code blocks which have to be executed as atomic transactions. Then the underling STM layer takes care of all the synchronization issues providing the illusion that transactions are executed serially. So the programmers can reason serially about the correctness of their applications. The STM layer, of course, doesn't execute the transactions serially. Actually, hiding all synchronization issues to the application, it allows multiple transactions to execute concurrently by relying on a Concurrency Control Protocol (CCP).

We stated that Transactional Memories are inspired to transactional DBS, but between them exists some basic differences [22]. The execution of in memory transaction ensures atomicity, isolation and consistency. Durability instead is not ensured because all the read and write operations are executed only in volatile memory. This brings to another significant difference: the execution time is usually smaller compared to database transactions because the access to persistent storage during data update is not necessary. Moreover in memory transactions don't pay the cost of the overheads for SQL parsing and plan optimization that characterize database environments. This differences allow STMs to ensure transactions execution times usually two or three orders of magnitude smaller than in conventional database environments [23, 24]. Another difference between STM and DBS is related to the isolation level required for memory transactions. For DBS serializability is considered largely sufficient, but this level of isolation allows transactions that will be subsequently aborted to read inconsistent data. In [25] the authors show that the effects of observing inconsistent states can have much more negative side effects in STMs that in DBS. In fact in STMs transactions can be used to manipulate program variables that directly affect the execution flow of user applications so, due to the observation of arbitrarily inconsistent memory states could bring the application to stall in infinite loops or in exceptions that may never occur in any sequential execution. Instead for DBS transactions are executed via interfaces with defined and more restricted semantic (e.g. SQL interfaces) and are executed in a "bulletproof" component, the database management system (DBMS), designed to avoid crashes or hangs in the case the transactions observe inconsistent data. For these reasons in memory transactions [26] require an isolation level called opacity, higher than serializability, that in addition prevents all transactions (also transactions that will be subsequently aborted) from seeing inconsistent values of data items. Today research on STMs topics is very active. Commercial releases of STMs do not exist yet but many research prototype (e.g [6, 19, 27, 28]) and prototype for commercial systems (e.g. [29]) are available. Moreover Intel recently released a processor with Transactional Synchronization Extensions [30] that represents the first (low cost) commercial implementation of hardware transactional memory.

2.2 The problem of concurrency in STM

One of the main challenge that a programmer have to face when designing parallel and distributed application is scalability, that is the ability of an application to proportionally increase its performance when the amount of available computing resources is increased. Focusing our attention on centralized multi-core systems, an ideal parallel application should scale linearly with the number of available core. Usually this level of scalability is not reachable due to two main factors:

- contention on physical resource,
- contention on logical resource.

Whit physical resource contention we denote the contention experienced by processes/threads that compose the parallel application when they try to access shared hardware resource (e.g. memory buses). It is strictly related to the specific platform used to run the application and keeping under control its side effect on performance require a detailed knowledge of the hardware. With logical resource contention we denote the contention experienced by the threads that compose the parallel application when they try to concurrently access shared logical resources (e.g. data in main memory). It is strictly related to the application logic and usually its side effects on application performance are higher than the ones due to physical contention: it brings to an higher performance degradation and it limits the scalability more than physical contention that can become negligible. To limit the impact of logical contention on scalability and performance the programmer must have a detailed knowledge of the application logic and of the application data access pattern. In this way the programmer can:

- divide properly the work between threads: the programmer should try to divide the work in a way that the needed synchronization between process/thread is minimum;

- when synchronization between thread/process is not avoidable: the programmer should use synchronization mechanism that minimizes wasted time.

But as we explained in paragraph 2.1, in the context of centralized transactional applications, these tasks are not trivial and STM helps to simplify developer's work taking care of synchronization of access to shared data. This implies that the logical contention experimented by the transactional application is closely related to the conflict detection mechanism implemented inside the STM. More in detail the performance of STM based application depends essentially on three factors:

- the specific transactional application workload,
- the conflict detection and contention management mechanisms implemented inside the STM,
- the level of parallelism used to execute the application.

About transactional application workload, two of the main parameters that can characterize a transactional workload are:

- the ratio between read and write on shared data,
- the distributions of read and write operation on shared data.

Workload with an high shared read/shared write ratio can be defined read intensive. Given an enough big fixed dataset, this type of workload, if the read and write operations are not concentrated on few shared data, usually brings to a low logical contention having, in this case, a little impact on application performance. Workload with low shared read/shared write ratio can be defined write intensive. In this case, especially if the read and write operations are concentrated on few shared data, the logical contention is high and has a significant impact on application performance. Between these two extremes, the logical contention can vary proportionally as a function of the ratio between shared read/shared write ratio and the shared data access distribution.

The conflict detection mechanism determines the point at which an inconsistency in the shared object is detected. Different design choices can have a substantial impact on the performance of an STM platform and on the degree to which it is suited to different kinds of workload. So it is necessary to find the right trade-off between the design choices that avoid wasted work (due to transaction abort) and the ones that avoid concurrency loss (where a transaction is stalled or aborted, even though it would eventually commit). Three main conflict detection mechanism families can be defined:

- *eager*: also called pessimistic, with this type of policy a conflict is detected as soon as it arises, aborting transaction immediately without waiting for the commit stage;
- *lazy*: also called optimistic, with this type of policy a conflict is detected at commit time, so the transaction is executed until it reaches the commit and then eventually aborted;
- *mixed*: this policy is a mix of the previous ones, it detects write-write conflicts early (since at most one of the conflicting transactions can ever commit) and it detects read-write conflicts late (since both may commit if the reader does so first).

In [31] a performance comparison using experimental evaluation of benchmarks between eager and lazy conflict detections have been presented. The results shows that in those applications where threads share a small number of data, the lazy policies are faster than the eager ones, while the latter are better in applications where threads share several data among themselves. This result brings to the conclusion that no single policy is superior across all kinds of workloads.

A Contention Manager(CM) that implements one or more contention resolution policies can be used to mitigate the side effect on performance due to conflicts between transactions. When a conflict is detected the actions taken by the CM to resolve it depend on the specific implemented resolution policy that can select whether:

- to abort the transaction t_1 that detects the conflict or
- to abort the opponent transaction t_2 that it encounters or
- to delay or not either transactions.

A lot of different contention management policies have been developed [32, 33]:

- *Passive*: t_1 aborts and re-executes
- *Aggressive*: each opponent transaction t_2 is immediately aborted
- *Polite*: for a fixed number N of exponentially growing interval of time, t_1 waits that the opposite transaction t_2 commits. At the end of each interval t_1 checks if the conflicting transaction has finished with the data. If the check fails for all the N intervals than the opposite transaction t_2 is aborted.
- *Karma*: this is a priority based policy in which an acquiring transaction immediately aborts other conflicting transactions with lower priority. If

the acquiring transaction t_1 has lower priority it wait trying to acquire the access to the data for N times, where N is the difference between the priority of the conflicting transactions. If the opponent transaction t_2 has not completed after all the N iterations, it is aborted. The priority of each transaction is established on the basis of the number of data accessed by the transaction.

- *Eruption*: this policy is similar to Karma but it adds the blocked transaction's priority to the active transaction's priority.
- *Greedy*: Each transaction obtains a timestamp when it starts the first time. If a conflict occurs t_1 aborts t_2 only if the timestamp associated to t_2 is higher to the ones associated to t_1 or if t_2 is waiting. Otherwise t_1 starts waiting for t_2 indefinitely.
- *Kindergarten*: in this policy a transaction t_1 is aborted each time that it conflicts for the first time with another transaction t_2. If t_1 conflict again, one or more time, with t_2, then t_2 is aborted.
- *Polka*: this policy is a combination of Karma and Polite. In few words Karma is modified introducing the polite's exponential backoff for the N waiting intervals.
- *Timestamp*: each transaction opponent t_2 is aborted if it started is execution after t_1.
- *Published Timestamp*: is similar to timestamp. It aborts older transaction that appear inactive, too.
- *Priority*: this policy aborts immediately the younger of the conflicting transactions.

In [32] a performance comparison using experimental evaluation of benchmarks between most of the previously illustrated policies have been presented. The authors state that Priority and Greedy policies provide the best performance overall, depending on the specific benchmark. Polka is still competitive but it can't reach the same performance of Priority and Greedy. The authors shows that all delay-based CMs (which pause a transaction for finite duration upon conflict) are unsuitable for the evaluated benchmarks even with moderate amounts of contention. However the results bring again to the conclusion that no single contention management policy is superior across all kinds of workloads.

The last factor that can affect transactional application performance is the level of parallelism used to run the application. When this level is too high a loss of performance may occur due to excessive data contention and consequent transaction aborts. Conversely, if concurrency is too low, the performance may be penalized due to limitation of both parallelism and exploitation of available resources. More in detail, given a specific transactional workload (that is a specific application) and a STM implementation with its specific concurrency control algorithm, maintaining constant the application load, the concurrency level affects the application performance in this way:

- starting from the sequential execution, increasing the level of parallelism the performance improves. The logical contention and the hardware contention increases, too. The increase of the contention brings to the increasing of the number of transaction abort due to conflicts on shared data. The transaction aborts brings to wasted time. But the number of transaction aborts is not high enough to override the gain due to the increasing level of parallelism. This performance gain occurs until a maximum is reached.
- starting from the level of parallelism that ensures the best performance,

the logical contention increases so much that brings the application to be subject to a so high number of transaction aborts that bring to waste so much time that the gain due to increased parallelism is not enough to compensate the wasted time. So the performance start to get worse.

To verify this behaviour we make some experiments executing STAMP, a very well known benchmark suite for STM that contains 8 different applications, on top of a STM implementation called Tiny-STM [19]. For our experiments we used an HP proliant server equipped with two AMD Opteron^{TM}6128 Series Processor, each one having eight hardware cores (for a total of 16 cores), and 32 GB RAM, running a Linux Debian distribution with kernel version 2.7.32-5-amd64 (This hardware and software platform is the reference for all the thesis, so in the next chapters, if not explicitly pointed out differently, we will always make reference to this platform). The results are showed by the graphs in Figure 2.1. The first graph shows the total execution time for the intruder benchmark varying the level of parallelism. As we can see the performance increase until 5 parallel threads. Beyond this level, the number of transaction abort increase so much that the total application execution time starts to grow. A very similar behaviour can be seen in all the other graphs except the last one. All the graphs shows performance that initially increases until it reaches an optimum and then, increasing the level of parallelism behind the optimum, the performance starts to decrease. The main difference between them is the optimal concurrency level that is strictly related to the application workload. The last graph shows the performance for ssca2 benchmark. We can see that it is different from the other ones. This is due to the specific workload of ssca2 that presents a very low logical contention: in the available hardware there aren't enough cores to reach a level of parallelism that produces an amount of logical contention sufficient

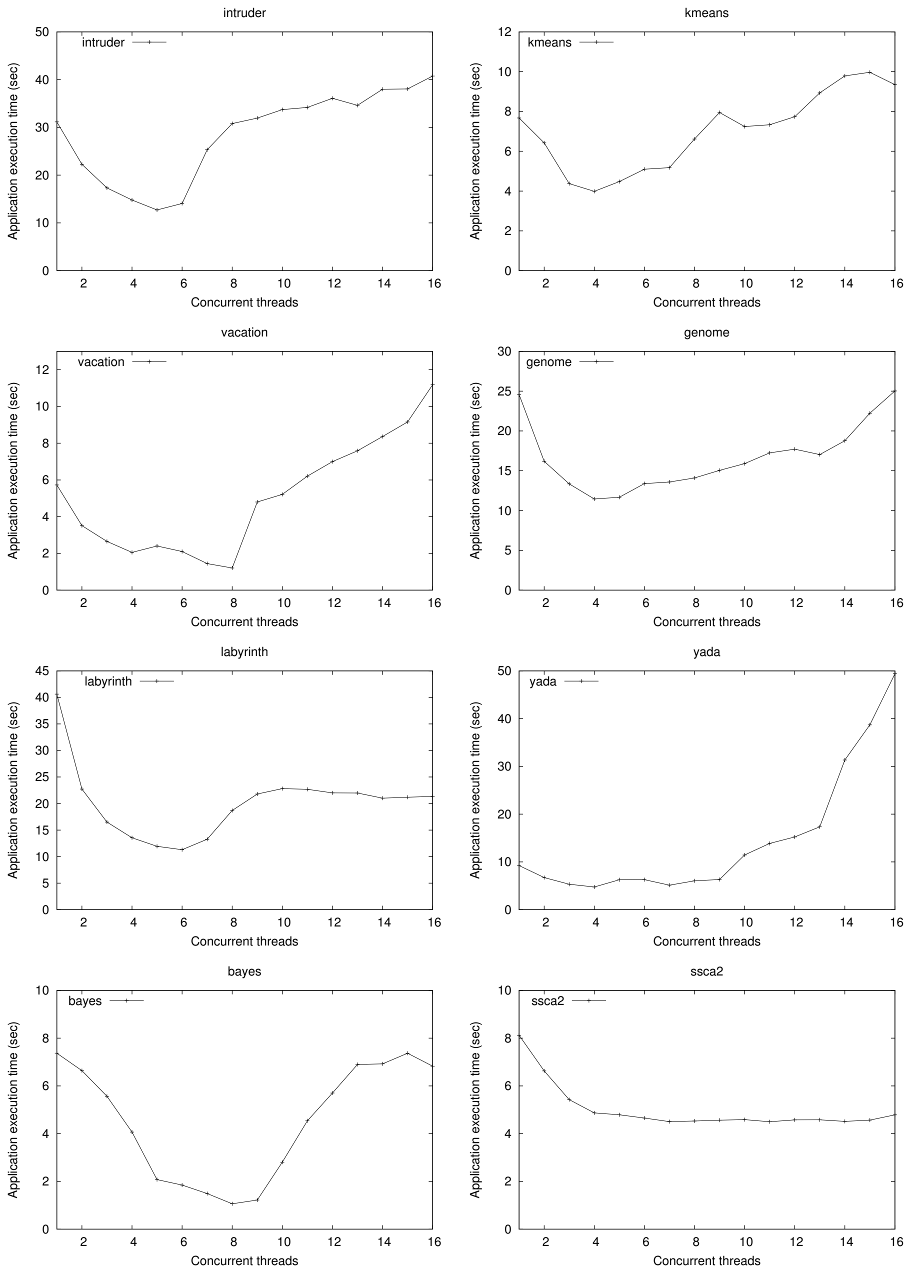


Figure 2.1: Total application execution time varying the concurrency level

to penalize the application performance. Similar results were shown in [34] for some application of the STAMP benchmark suite not only for TinySTM but for other STM implementations, namely SwissTM [35] and TL2 [6], and in [36] for different versions of SwissTM.

To limit the side effect of logical contention and improve/optimize the performance, three different methods can be used:

- implement better concurrency control mechanisms: one way to limit the side effect of logical contention is develop efficient concurrency control mechanisms that are able to properly synchronize the shared data access in a way that allows to minimize the wasted time due to transaction aborts. As we will see in the chapter 3, a lot of concurrency control mechanisms have been developed in literature. Unfortunately no one of them is better than the others: usually each concurrency control mechanism is optimized for a specific subset of workloads, for which it provide optimal performance. With workloads different from those for which they are optimized, they are not able to provide the same optimal performance;

- implement transactions scheduling mechanisms that are able to detect high conflict execution interval and to take scheduling decision, like for example serialization of high conflicting transactions, to limit the side effects of logical contention;

- implement mechanisms that are able do detect the optimal level of parallelism (that is the one that ensure the better trade off between the loss due to transactions rollback and the gain due to increased parallelism). Different type of mechanisms have been developed. As best of our knowledge the implemented solutions are based on:

- analytical model
- heuristics
- machine learning algorithms

We developed a suite of solutions, one based on an analytical model, one based on machine learning and a hybrid solution that combines analytical modelling and machine learning. All the approaches are general, they can be used with all transactional application types and with all the STM implementations, independently of the concurrency control algorithm used inside the STM framework. These approaches are orthogonal to scheduling and concurrency control mechanisms and can be used in combination with them. The hybrid approach is particularly interesting because it combines machine learning with analytical modelling techniques taking the best from the two approaches.

CHAPTER 3

Performance Optimization in STM, State of Art

An effective way to face performance tuning and dynamic resource allocation problems is to develop a mechanism that allows to dynamically choose the better system configuration on the basis of the current system input. This mechanism, to make the right choices, should exploit some kind of model that allows to predict the performance varying the system input and configuration. The methodologies to develop a model like this can be grouped within two main classes:

- **White box approach**: in this class fall all techniques that require a detailed knowledge of internal system dynamics. That is, it is necessary a detailed study of the system to identify all its fundamental characteristics and the relations among them. As the name suggests, this methodology looks inside the system to deeply understand the factors that determine its behavior. This type of approach allows to obtain a model with good extrapolation power, that allows forecasting system behavior in unexplored regions of its parameters space. It also requires minimal learning time because it is sufficient to instantiate some basic model parameters. The main disadvantage of this approach is that the analytical model is complex and expensive to design and validate, the complexity of the model implementation grows with the complexity of the system. Moreover, the mathematical model is an approximation of the real system behavior, so it should be subject to approximation errors.

- **Black box approach**: this approach is the exact opposite of the previous one. Black box techniques observe only inputs, context and outputs of the system and use statistical methods to identify internal system's patterns and rules. This type of approach allows to obtain good accuracy in already explored regions of the parameters space but usually it doesn't ensure good extrapolation power. Black box techniques require learning time that can grows in a not acceptable rate with the number of system features, but in most cases they eventually outperforms analytical models. Examples of black box approaches are the Machine Learning techniques: the system model is developed using a methodology that observe data representing the system behaviour. That data can be incomplete, that is they can represent system behavior only in some working area. Machine Learning algorithms allow the model to learn via inductive inference based on observing data that represents incomplete information about statistical phenomenon and generalize it to rules and make prediction on missing attributes or future data.

It is possible to combine techniques that belong to white box approach with techniques that belong to black box one. In that way we can obtain hybrid solution (**Grey box approach**), that allows to exploit the strengths of both the approaches (e.g. Machine Learning techniques can be combined with analytical modeling [37]).

With the words Machine Learning we denote a set of learning methodologies based on data and past experience that can be classified in three main classes:

- *supervised approach*: learning data are represented by pairs $(\mathbf{in}, \mathbf{out})$, which represent the output of the system given a specific input. Some examples of supervised approach are: Decision Trees[38, 39, 40], Neural networks[38], Support Vector Machine[41], that can be classified as off-line techniques, too;

- *unsupervised approach*: the learning data don't denote the outputs obtained by the available inputs. It brings to bear prior biases as to what aspects of the structure of the input should be captured in the output. A detailed description of unsupervised approaches can be found in [42];

- *reinforcement learning approach*: just like in unsupervised approach the outputs are not available, but the algorithm can measure a delayed reward. An example of reinforcement learning technique is the UCB algorithm[43] (Multi-armed bandit problem [44, 45, 46, 47]) that is an on-line technique.

Machine Learning techniques are typical black box approaches. They can reach good accuracy in already explored regions of the parameters space, that is regions covered by learning data, but they can't always ensure good extrapolation power. This problem is usually referred as over-fitting: the algorithm identifies a model that approximates very well the learning data, but it has poor generalization performance. As previously stated in general for black box approach, another disadvantage of Machine Learning techniques is that the learning time can grow up in a not acceptable way with the number of system's input/output parameters and the size of learning dataset. But if we control the over-fitting problem and if we can spent some time in training task, Machine Learning techniques can outperform analytical models.

In contexts different from STM, some performance optimization approaches based on concurrency regulation have been proposed.

In [48] the authors propose to adapt the concurrency level of parallelizable portions of a scientific (not transactional) application. In this approach the concurrency level is determined before the start of each parallel code portion and it is not possible to change the concurrency level during the execution of the code portion. To adapt the concurrency level they propose three different strategies:

- *speedup-driven incremental search strategy* (SISS), that is a hill climbing technique that uses thresholds to identify the optimal concurrency level;
- *speedup-driven global search strategy* (SGSS), that periodically executes a comprehensive search evaluating the performance for all the possible concurrency level (from 1 to the maximum number of available processors) and then it chooses the level with the highest speed-up under the restriction that the efficiency has to be above a given threshold value;
- *efficency-driven global search strategy* (EGSS), that is a variation of the previous one that chooses the setting with the highest efficiency under the constraint that it must guarantee a speed-up of at least a certain threshold value;

In [49] the authors study the tuning of the concurrency level (number of concurrently running transactions) within a transactional processing system (e.g. database server) running on a single machine with the aim to avoid thrashing phenomena due to system overload. They propose two algorithms:

- the first one is a hill climbing approach, that starts the execution with and arbitrary number n of concurrent running transactions and then it update n by one at each time step measuring the resulting performance. If it decreases the algorithm turn the updating direction of n until again the performance becomes worse, and so on. The updating process of n works in a zig-zag fashion;

- the second one uses a second-degree polynomial function to model the relation between the workload and the system performance. Than it uses this polynomial model to take decision about the concurrency level to use for application execution.

In [50] the problem of regulating the multiprogramming level of a database server to improve its performance is faced using a feedback control loop that is initialized with a close-to-optimal value thanks to the use of queueing theoretic models. The usage of queueing models allows the approach to converge fast under abrupt workload changes, too.

In [51] the authors propose a hybrid approach merging the previous two solutions.

3.1 Approaches for performance optimization in STM

As already stated, one core issue to cope with in STM is related to exploiting parallelism while also avoiding thrashing phenomena due to excessive transaction rollbacks, caused by excessive contention on logical resources. In order to deal with this issue, several literature proposals exist. These approaches can be grouped within different sets of orthogonal solutions. On one hand we can find optimized schemes for transaction conflict detection and management [6, 7, 8, 9, 10]. These include proposals aimed at dynamically determining which threads need to execute specific transactions, so to allow transactions to be expected to access the same data to be run along a same thread in order to sequentialize and spare them from incurring the risk of being aborted with high probability. Two interesting works are [11, 52] where Machine Learning techniques are used to select the best performing conflict detection and management algorithm depending on the specific application workload.

Other proposals rely instead on pro-active transaction scheduling [12, 13] where the reduction of performance degradation due to transaction aborts is achieved by avoiding to schedule (hence delaying the scheduling of) the execution of transactions whose associated conflict probability is estimated to be high. All the above schemes are not meant to optimize the number of threads to be used for running the application, thus they generally operate on top of configurations where the number of threads is predetermined at application startup.

In the approach proposed in [12], incoming transactions are enqueued and sequentialized when an indicator, referred to as *contention intensity* CI, exceeds a pre-established threshold. The contention intensity is calculated, by each concurrent thread, as a dynamic average depending on the number of aborted vs. committed transactions. That approach proposes a technique, called adaptive transaction scheduling (ATS), where an adaptive scheduler controls the number of concurrent transaction that can access to critical section on the basis of contention feedback coming from the application. This is done through the selectively scheduling of transactions subject to frequent aborts. The scheduling scheme specifically deals with when to resume the aborted transaction: it dynamically chooses the point where an aborted transaction must resumes its execution. When a thread starts to execute a transaction or resumes a transaction after an abort it evaluates its contention intensity CI: initially the contention intensity is set to 0 and then, at each evaluation step, it is evaluated as $CI_n = \alpha \times CI_{n-1} + (1 - \alpha) \times CC$ where CI_n and CI_{n-1} are respectively the contention intensity at evaluation n and at evaluation $n-1$, α is the weight variable, and CC is the current contention that has value 0 if the transaction commits or 1 if the transaction aborts. After the evaluation of the contention intensity each thread compares its CI with a designed threshold. When it is below the threshold, the thread begins a transaction normally. Otherwise, the thread will report to the scheduler and it will start waiting for a dispatch. The scheduler will take decision about the time t to execute the transaction and then signal back the thread to proceed when t arrives. The scheduler maintains a single centralized queue of transactions used to dispatch one single transaction at a time: a new transaction from the head of the queue is dispatched for the execution only when the previously selected transaction from the queue executes a commit or an abort. Note that this queuing behaviour effectively serializes high contention transactions. In this approach the scheduler doesn't take scheduling decision for all the executed transactions but only for those that start under high contention. This infrequent access allows the scheduler to be implemented as a centralized module, thereby enabling an advanced and coherent system-wide scheduling scheme. Such approach doesn't affect negatively the performance when the contention is low and it limits the performance degradation when the contention grows.

In the proposal presented in [13], a transaction is sequentialized when a potential conflict with other running transactions is predicted. The prediction leverages on the estimation of the expected transaction read-set and write-set (on the basis of the past behaviour of other or the same transaction). Actually, the sequentializing mechanism is activated only when the amount of aborted vs. committed transactions exceeds a given threshold. More in detail, in this work the authors introduce a scheduler, called Shrink, which predicts the future accesses of a thread on the basis of the past accesses and dynamically serializes transactions based on the predictions to prevent conflicts. The scheduler is based on two main ideas: *locality of reference* and *serialization affinity*. The locality is used in two forms:

- To make prediction about the transactional read sets, the notion of temporal locality [53, 54] is used. In the context of transactional memory temporal locality means that the address frequently accessed in the past from last transactions executed by a thread are more likely to be accessed in future transaction of that thread. The scheduler, for each thread that execute transactions, maintains the read set of the past few committed transactions. Then, using a confidence measure to predict if the address could be read in future transactions, the scheduler checks the membership of an address in these read sets. To predict transactional write sets the scheduler uses the locality across repeated transactions. To prevent conflicts than scheduler uses the information (about read and write operation) from the currently executing transactions in conjunction with the predicted accesses sets. More in detail, given a transaction t_i that is just starting his execution, the scheduler compares any addresses in the predicted read set and write set of t_i with the addresses written by any other currently executing transaction t_j. If at least a comparison is positive, Shrink serializes the starting transaction, otherwise the transaction executes normally.

- to avoid performance degradation in low contention cases the serialization is used only if the contention is high. To understand when a thread is subject to high contention, the scheduler maintains a parameter for each thread called success rate and activates the prediction and serialization techniques for a thread only if its success rate goes under a certain threshold. To establish the threshold, an heuristic called serialization affinity is used: serializing a transaction is more helpful when a large number of threads access similar addressed and compete for a small number of cores. So, on the basis of this heuristic, the scheduler serialize a transaction with probability proportional to the contention in the TM.

The authors state that the scheduler can be integrated with any STM that uses visible writes (e.g. [35, 55, 56])

Another set of solutions is aimed at supporting performance optimization via the determination of the best concurrency level (number of threads) to be exploited for running the application on top of the STM layer. These solutions are orthogonal to the aforementioned ones, being potentially usable in combination. We can further distinguish these approaches depending on whether they can cope with dynamic or static application execution profiles, and on the type of methodology used to predict the optimal level of concurrency for a specific (phase of the execution of the) application.

Approaches coping with static workload profiles and using analytical modelling techniques for performance prediction are [16, 17]. This approaches are not able to predict the optimal level of concurrency for applications which generate dynamic workloads that evolve over time.

The work in [16] presents an analytical model taking as input a workload characterization of the application expressed in terms of transaction profiles (length of transactions, transactions arrival frequency, number of checkpoints and computing cost of transactions), contention probability and hardware resources consumption. The model predicts the application execution time (estimating the wasted time due to conflicts) as function of the number of concurrent threads sustaining the application, however the prediction is a representation of the average system behaviour over the whole lifetime of the application. The model is based on queuing theory and each transaction is modelled as a client that request services from the computing system. To describe the start and the completion (commit or abort) of the transaction a continuous time Markov chain is used. In this approach the input parameters need to be calculated by running the application and profiling the workload, including measurements of the transaction conflict probability, and by inspecting the application code. Predictions are related to the execution scenario of the profiled application as determined by the workload configuration used to run the application. Hence, changing the workload configuration of the application, a new profiling may be required. In addition, predictions are related to the entire execution of the application. Because during the lifetime of an application some features, as the workload profile and the transaction conflict probability, can change, then it is not possible to perform dynamic predictions on basis of the current workload of the application.

The proposal in [17] is targeted at evaluating scalability aspects of STM systems. It relies on the usage of different types of functions (such as polynomial, rational and logarithmic functions) to approximate the performance of the application when considering different amounts of concurrent threads. The approximation process is based on measuring the speed-up of the application over a set of runs, each one executed with a different number of concurrent threads, and then on calculating the proper function parameters by interpolating the measurements, so as to generate the final function used to predict the speed-up of the application vs the number of threads. More in detail, approximation techniques [57] are used to predict the performance of a workload using n threads, based on its performance with k_i thread, with $1 \leq i \leq K$. An analytical performance prediction function $f(n)$ based on m measurements describes the characteristics of the application workload and of the underling computing environment (hardware, operating system, STM framework). The function takes as input parameter the number of thread n and gives as output an estimation of the expected performance. The approach consists of three main steps:

- a profiling step in which the workload performance is measured with several thread counts, obtaining a set of measures.
- an interpolation step in which the the collected measures are used to build a performance function $f(n)$
- a prediction step in which the function $f(n)$ is used to predict the application performance with number of threads different that the ones used during the profiling step.

This approach doesn't require any knowledge of the system and of the workload when constructing f (no access to the source code of the application is required) but it has a limitation due to the fact that the workload profile of the application is not taken into account. Hence the prediction may prove unreliable when the profile gets changed wrt the one used during measurement and interpolation phases. If it changes, e.g. in terms of transaction profiles, over the lifetime of the application, the performance achieved with a given number of concurrent threads can change. As a consequence, the calculated performance function may become unreliable, unless calculating it again by taking new measurements.

Other concurrency level optimization approaches that have been proposed in literature relying on analytical methods for performance prediction are [14, 58, 59, 60].

In [14] an analytical modeling approach that captures dynamics related to the execution of both transactional read/write memory access and non-transactional operation has been proposed. The model is used to evaluate the performance of STM applications as a function of the number of concurrent threads and other workload configuration parameters (E.g. execution cost of transactional and not-transactional operations, cost of begin, commit and abort operations). This kind of approach is targeted at building mathematical tools allowing the analysis of the effects of the contention management scheme on performance and it is based on a two-layered analytical modelling methodology:

- independently of the specific scheme used for regulating memory access by concurrent transactions, a thread-level model is used to predict the system performance as a function of:
  - the number of worker threads that execute transactional memory operation,
  - the probability that they are executing non-transactional or transactional code blocks;
- a transaction level model, that can be specialized for a given concurrency control scheme, is used to determine commit/abort probabilities on the basis of the specific choices determining the actual synchronization scheme

among threads executing conflicting transactional code blocks.

To develop that models a detailed knowledge of the specific conflict detection and management scheme used by the target STM is required.

In [58] the authors propose a formal model of transactional memory performance called Syncchar. Their approach starts running a lock-based or transactional parallel application and than it samples the addresses written and read during critical sections. Then a model of the program's execution is built and it is used to predict the performance of the application if it uses transactions. This model uses two metrics: *data independence* and *conflict density* of the critical sections. The first metric measures the likelihood that threads will access disjoint data. The second metric measures how many threads are likely to be involved in a data conflict that should occur.

In [59, 60] a set of models for different type of STM has been proposed. The authors consider the behaviour of a representative transaction, called tagged transactions, whose execution is influenced by the side-effects of other concurrent transactions. They formulate the influence that these transactions exercise on the tagged one by defining some aggregate parameters representing the mean numbers of transactional data held in shared or exclusive state and the conflict probability. They models the STM dynamics at each read/write and at the commit point of the tagged transaction using an absorbing discrete-time Markov chain (DTMC). Then with this model they derive expressions for the conflict probability, the average number of data held in exclusive/shared state and the average number of requests issued in exclusive/shared data until completion of the tagged transaction.

Other concurrency optimization approaches relying on black-box Machine Learning (ML) methodologies are [61, 15, 62].

In [61, 15] an exploration-based approach that periodically performs on-line monitoring of the number of transaction commits and aborts and then decides to increase or decrease the level of parallelism has been developed (hill climbing technique maximizing transaction commit rate). For Distributed Transactional Memories (DTM) two approaches have been developed: transactional auto scaler (TAS) and self correcting transactional auto scaler (SC-TAS). TAS [62, 15] relies on a mixed Machine Learning/analytical modelling(AM) approach in which the AM is used to capture the data contention dynamics and the ML is used to predict the inter-node communication latencies in a DTM platform. The advantage of using ML lies in its black-box nature, which makes it a very well-fitting choice for coping with performance forecasting of components in cloud infrastructures, where typically there is little knowledge of the hardware system architecture, particularly as concerns the network. SC-TAS [15] extends TAS exploiting the idea of learning, by means of on-line ML techniques, a correction function to the output of TAS, hence allowing to minimize the prediction errors of TAS AM-based forecaster.

Another approach for concurrency level optimization that doesn't use neither Machine Learning nor analytical modeling is [63]. In this approach a control algorithm dynamically changes the number of threads which can concurrently execute transactions on basis of the observed transaction conflict rate. It is decreased when rate exceeds an threshold while it is incremented when the rate is lower than another threshold.

The last two works that we report are [64, 65]. They can not be included in the previously discussed classes of solutions, but they are however aimed to optimize the performance of STM based applications. In these proposals Machine Learning is used to select the most suitable thread mapping, i.e. the placement of application threads on different CPU-cores, in order to get the best performance.

Analysing the various reported approaches we can see that between all the proposed optimized schemes for transaction conflicts detection and management none of them results to be better than the others: experimental results already presented in literature showed that each approach can overcame the others only for a specific subset of all the possible transactional workload profiles. Moreover, as we can see from the same experimental results, the performance of all these approaches depends on the concurrency level used to execute the application. As a consequence, running the application with a non optimal concurrency level can bring to performance loss, independently of the used transaction conflict detection and management mechanism. Similar considerations can be done for the transactions scheduling and cpu-thread mapping approaches. None of them optimize the concurrency level and for this reason they are prone to performance degradation when a not optimal number of threads is used to execute the application.

So, dynamic concurrency regulation turns out to be an essential building block to be exploited with the aim of obtaining the maximum performance from the previously discussed approaches. Being totally orthogonal to them, the techniques aimed at support performance optimization via the determination of the best suited number of threads used to execute the application can be used in conjunction with the other solutions to obtain best performance form STM based applications.

All the already proposed approaches for concurrency level optimization present some significant drawbacks. Starting from the ones based on analytical modelling we can see that the approaches in [16, 17] cope with static workload profiles, so they are not able to predict the optimal concurrency level for applications characterized by dynamic workloads that evolve over time. The approaches in [14, 58, 59, 60] are better than the previous ones but they propose very complex models that require a detailed knowledge of the internal mechanism of the software transactional memory to be instantiated. The proposals in [61, 15, 62] use machine learning and are essentially based on hill climbing techniques. Such approaches can be very slow to converge (they can require a not negligible time to find the optimal concurrency level) and they can be non-reactive with applications that present a transactional workload profile that varies rapidly.

In this thesis we propose a suite of solutions for dynamic concurrency regulation that allows to overcome the just mentioned drawbacks. We developed three different approaches: one based on machine learning, one based on analytical modelling and a hybrid solution that combines the previous two (allowing to chase the best of both the methodologies by tackling the shortcomings intrinsic in each of them). All the proposed approaches are general, they can be used with all transactional application types and with all the STM implementations, regardless of the concurrency control algorithm used inside the STM framework. Moreover, they are orthogonal to scheduling and concurrency control mechanisms and they can be used in combination with them. They are very simple to instantiate, that is they don't require a detailed knowledge of the internal STM operating mechanism, and they are able to cope with cases where the actual application execution profile can change (also rapidly) over time. Furthermore, as we will show later in this thesis, the hybrid approach represents a methodology that allows to build very quickly a highly reliable performance model allowing the determination of the optimal level of concurrency for a specific STM based application. Moreover, to reduce the applications monitoring overhead (that can penalize the performance) without affecting the accuracy of the predictors used in our approaches we developed a fully innovative mechanism that allows to dynamically select the composition of the input features set exploited by the performance prediction. Summarizing, we provide a lightweight and complete framework for concurrency regulation in STM based applications that allows to obtain always optimal performance independently from the specific STM implementation and hardware used to execute the application.

Chapter 4

Application Model and Workload Features

In this section we depict the model of the STM application. After, we discuss the application performance model we exploit in our approaches, also describing motivations associated to the choice of parameters we use to characterize the application workload and their expected impact on the system performance.

4.1 Model of the STM application

We consider an application executing with a number of concurrent threads that can be activated and deactivated in order to optimize the concurrency level. We denote with m the number of active threads running at a given point of the execution of the application. The execution flow of each thread is characterized by the interleaving of transactions and non-transactional code (*ntc*) blocks (i.e. code blocks outside of transactions). Any transaction starts with a *begin* operation and ends with a *commit* operation. During the execution of the transaction, a thread can both (A) perform read and write operations on shared data objects, and (B) execute code blocks where it does not accesses shared data objects (e.g. it accesses variables within its own stack). Each shared data object read (written) by a thread while executing a transaction is included in the transaction read-set (write-set). If a conflict between two concurrent transactions occurs then one them is aborted and re-started. Right after the thread commits a transaction, a *ntc* block starts, and it ends right before the execution of the *begin* operation of the subsequent transaction along the same thread.

4.2 STM application performance model

With respect to the case of applications with no data contention, a major challange when predicting/evaluating the performance of transactional applications is to estimate the transaction execution time (i.e. the elapsed time between the first execution of the *begin* operation of a transaction and the time when the transaction commits), which is also affected by the *wasted* time associated to the aborted executions of a transaction. The expected wasted time is hard to estimate, because it depends on a lot of factors, including both workload profile (as the length of transactions in terms of executed instructions and shared data object accessed by transactions), and run-time system parameters (as the number of concurrent threads and code processing speed). Additionally, the workload profile and some run-time system parameters could also change during the execution of the application, so that the wasted time could considerably change over time. The performance prediction technique we use in our approach relies on run-time observation of some parameters characterizing the execution of the application and exploits a performance model allowing to predict the average wasted time of transactions as a function of the observed parameters. The function we exploit as the fulcrum of performance prediction is the following:

$$w_{time} = f(rs_s, ws_s, rw_a, ww_a, t_t, ntc_t, k) \tag{4.1}$$

where w_{time} is the average wasted time of transactions, rs_s is the average read-set size of transactions, ws_s is the average write-set size of transactions, rw_a is an index providing an estimation that an object read by a transaction could also be written by another concurrent transaction, ww_a is an index providing an estimation that an object written by a transaction could also be written by another concurrent transaction, t_t is the average execution time of the committed transaction runs (i.e. the average execution time of the transaction runs which do not get aborted [1]), ntc_t is the average execution time of *ntc* blocks and k is the number of concurrently running threads. The choice of the parameters was made on the basis of the experience acquired in the last years by my research group [14]. In the next paragraph we will discuss with more detail each one of the parameters chosen to characterize the application workload, the existing relations between them and how they affect application performance.

4.3 Input parameters analysis

The choice of parameters is of primary importance because only choosing the right set it is possible to obtain good performance predictions. We started from a small set that include the ones that can have greater impact on abort probability (and accordingly on wasted time) and than we enriched the set with additional parameters when the ones already identified were not sufficient to obtain good predictions. As already stated in section 4.2, the full set of parameters that can be used to characterize the transactional workload is composed by six elements:

[1]note that, according to the application model, a transaction, possibly after experienced a number of aborted runs, eventually experiences a committed run.

- *Mean committed transaction execution time (t_t)*: It is the mean time that a committed transaction spent to execute all the operations from the *begin* to the *commit*, without taking into account additional time due to aborts. Inside a software transactional memory the execution time Ex_{time} of a transaction is strictly related to the transaction abort probability P_{abort}: Ex_{time} is proportional to P_{abort}. Given a transaction t_1, if the mean execution time of t_1 grows, then the probability that another concurrent transaction t_2 access one or more item accessed by t_1 increase, too. t_t affects the abort probability and than it can affect the overall mean wasted time w_{time} due to transaction aborts. Moreover, when aborted, longer transactions entail longer wasted time with respect to shorter transactions, hence the average wasted time increases when the average execution time of the committed transaction runs increases.

- *Mean non-transactional execution time (ntc_t)*: It is the mean time that each thread spent to execute the non-transactional code from the *commit* of a transaction and the *begin* of the subsequent transaction. Differently from t_t, the non-transactional execution time is related to the abort probability in an inversely proportional way. Consider a thread *th*, if it is executing a non-transactional code block, than it for sure doesn't access shared data and than it can't be the cause of a transaction abort. This involve that more time *th* spends executing non-transactional segments, lower it will be the probability that *th* brings to the abort of other transactions executed by other parallel threads. Then, longest is the time that *th* spend to execute non-transactional code blocks, lower it will be the contribution to the global abort probability (of the whole system). So ntc_t, affecting the global abort probability, can affects the wasted time w_{time}, too. More simply longer execution time of ntc_t blocks determines a lower probability that two or more transactions are executed concurrently. As a consequence, we expect that the abort probability (thus also the wasted time) decreases when the execution time of ntc blocks increases.

- *Mean transactional read set size (rs_s)*: It is the mean size of the read set of a transaction. It affects in a proportional way the transaction abort probability. Intuitively, greater is the number of shared items that a transaction t_1 reads, grater is the probability that another concurrent transaction t_2 updates (at least) one of these items, bringing to the abort of t_1 transaction. Affecting the abort probability, the mean read set size can affect the amount of wasted time w_{time} due to transaction aborts.

- *Mean transactional write set size (ws_s)*: It is the mean size of the write set of a transaction. As for rs_s the mean write set size affects in a proportional way the transactions abort probability. In this case too, higher is the number of shared items that are modified by a transaction t_1, higher is the probability that another concurrent transaction t_2 will update (at least) one of the items modified by t_1, bringing to the abort of one of the two transaction. In this case too, affecting the abort probability, the mean transaction write set size can affect the amount of wasted time w_{time} due to transaction aborts.

- *Write-write conflict affinity index (ww_a)*: This index, that can assume values between 0 and 1, is obtained as the dot product between the vector of the distribution of the write accesses and himself. This index gives an estimation of the conflict rate between write operations of concurrent transactions. The higher the index value, the higher the conflict rate between writes is and so we expect an enhancement of the abort probability and of the wasted time.

- *Read-write conflict affinity index (rw_a)*: This index, that can assume values between 0 and 1, is obtained as the dot product between the vector of the distribution of the write accesses and the vector of the distribution of the read accesses. This index gives an estimation of the conflict rate between read and write operations of concurrent transactions. The relation between the wasted time and the values of this index is the same of the ww_a index.

The last parameter that affect the value of the function w_{time} is the concurrency level k. It represents the number of concurrent threads used to execute the application. This parameters doesn't characterize the transactional workload, but it directly affect the whole application performance: the number of concurrent transactions depends on the number of concurrent threads, hence the wasted time increases when k increases.

4.4 Input parameters - experimental data

In this paragraph we report some experimental data that confirm the correctness of our analysis. These data have been collected executing three applications belonging to the STAMP benchmark suite (kmeans, bayes and intruder) on top of the hardware platform already described in Chapter 2 using different level of parallelism. We show data related only to a subset of the applications available in STAMP because, as we will deeply discuss in Chapter 5, not always the full parameters set affect the total mean transactional execution time Ex_{time}. So we chose the applications for which the relation between the sampled parameters and Ex_{time} is more evident.

The graphs from 4.1 to 4.6 are obtained from the collected data, ordering the samples on the basis of the total mean execution time Ex_{time} (the mean transaction execution time including wasted time due to transaction aborts, obtained as the sum between r_t and w_{time}). They shows the relation between Ex_{time} and t_t (graph 4.1), the relation between Ex_{time} and ntc_t (graph 4.2), the relations between Ex_{time} and the parameters ws_s and rs_s (graphs 4.3 and 4.4) and the relations between Ex_{time} and the parameters ww_a and rw_a (graphs 4.5 and 4.6). As we can see the real data respect the behaviour previously described.

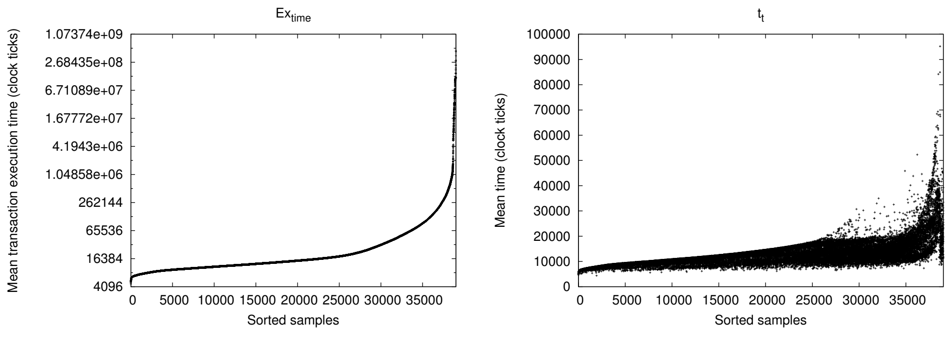


Figure 4.1: Total mean execution time vs. mean execution time

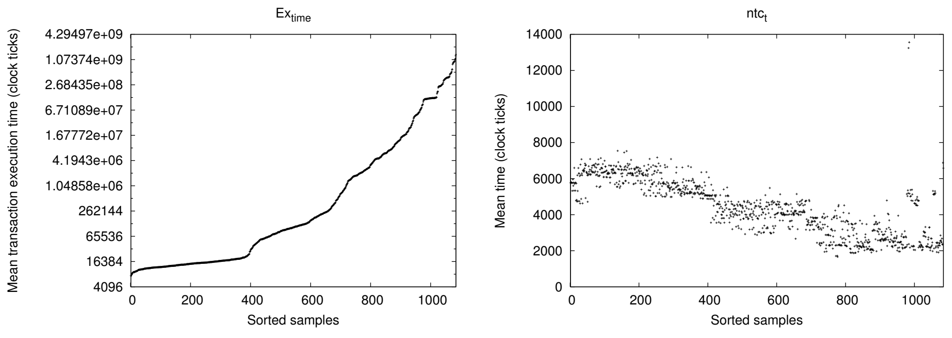


Figure 4.2: Total mean execution time vs. mean non-transactional time

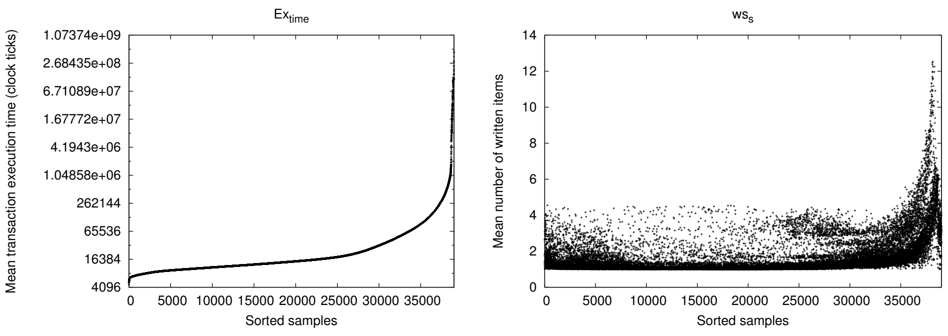


Figure 4.3: Total mean execution time vs. mean read write set size

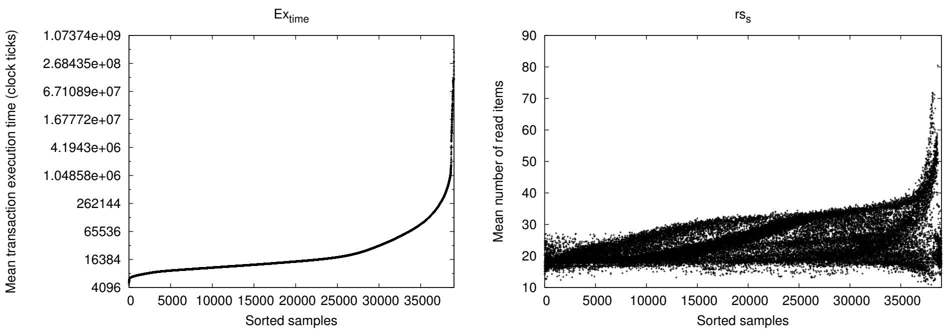


Figure 4.4: Total mean execution time vs. mean read set size

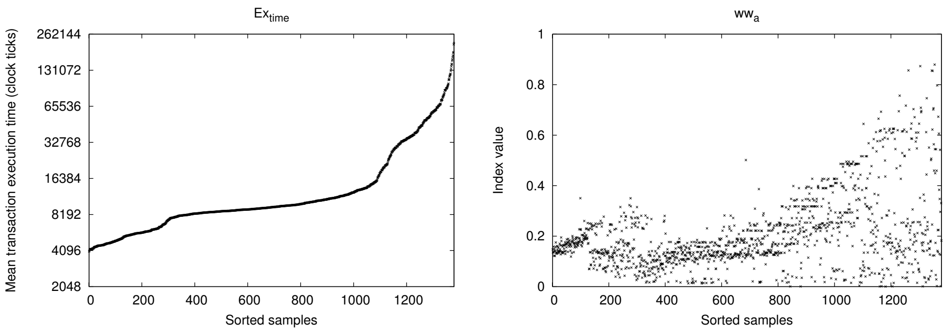


Figure 4.5: Total mean execution time vs. write-write index

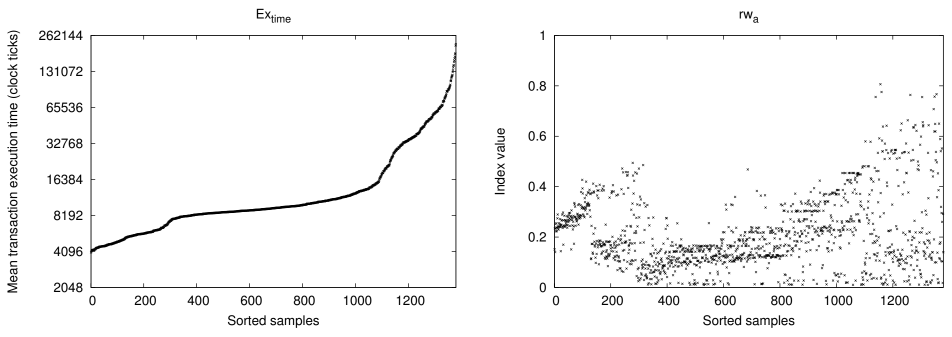


Figure 4.6: Total mean execution time vs. read-write index

Chapter 5

Machine Learning Based Approach

In this chapter we present a machine learning based approach for the concurrency regulation in STM. To solve this problem we need a mechanism that allow to predict the system performance varying the concurrency level used to run the transactional application. More in detail we have to develop a predictor that takes as input a statistical representation of the transactional workload and returns an estimation of the wasted time due to transaction aborts for each level of parallelism. Afterwards this estimation is used to predict the system throughput. In other words we need a predictor that gives a good approximation of the function 4.1 for each level of parallelism k, as already discussed in section 4.2. To do this, we have to fix a maximum level of parallelism as upper bound for the performance prediction process. This is a plausible hypothesis for two main reasons:

- the study in [66] shows that to obtain optimal performance with software transactional memory based applications the level of parallelism should be less or equal to the number of available cores;

- given a real parallel architecture, the number of cores that it can hold is finite.

So choosing the maximum value for k is an easy task: it is sufficient to know the number n of available cores of the reference hardware architecture and put $k = n$. After that we have to select the methodology for the prediction model development. We chose a black box approach, for the advantages already discussed in Section 3. The first step in model construction is the building of a set of samples, called training set, that allows to represent the function w_{time}. It will be used to train the performance predictor. This training set consists of a set of vectors, each one derived by observing the application during a given interval of time and including the following quantities (the apex t is used to indicate that the quantities are related to a training sample):

- the set of statistics that represent the transactional workload, i.e. rs^t_s, ws^t_s,rw^t_a, ww^t_a, t^t_t, ntc^t_t
- the average wasted transaction execution time w^t_{time}
- the number k^t of active threads (i.e the threads used to run the application during sample collection)

Hence, a training sample $(\mathbf{i}, \mathbf{o})$ is such that $\mathbf{o} = (w^t_{time})$ and $\mathbf{i} = (rs^t_s$, ws^t_s, t^t_t, ntc^t_t, rw^t_a, $ww^t_a, k^t)$. The features of the training samples together with other characteristics of the training set give to us some indications about the specific machine learning technique that can be used to learn the function f, more in detail:

- analyzing the structure of the training sample we can see that, given a configuration of parameters **i**, the sample contains the respective value for the output **o**. This structure of the training samples allows to restrict the choice to the supervised machine learning approaches;
- analyzing the elements inside the vector we can note that each of them is a positive real value;
- the software transactional memories based application has some instability, i.e. high variance of performance when the concurrency degree is higher than the optimal level (the variance usually grows with the distance from the optimal concurrency level). This can bring to have a high level of noise inside the training set;
- if all the input parameters are strictly necessary to obtain reliable performance predictions, then the function f that we must approximate can be very complex.

All this observations suggest to use Neural Networks (NN) [38] as the proper technique to approximate the function f, as we will show in paragraph 5.3. Once obtained a good predictor for the w_{time} function, it can be used inside a self-adjusting concurrency STM to take runtime decision about the level of parallelism to use during the application execution lifetime. In the next paragraph we will present an architecture that can be used to build a STM platform able to adapt the level of parallelism exploiting the output of the developed predictor.

5.1 System Architecture

The self-adjusting concurrency approach we propose leverages on three architectural building blocks, namely:

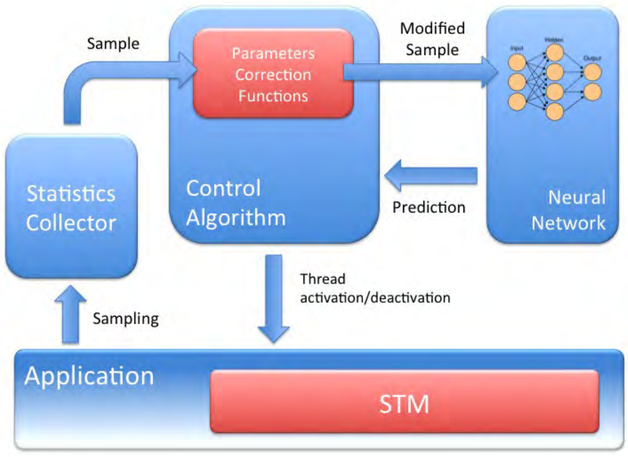


Figure 5.1: System Architecture.

- A Statistics Collector (SC);
- A Neural Network (NN);
- A Control Algorithm (CA).

The system architecture is depicted in Figure 5.1. When a workload sampling interval terminates (hence on a periodic basis) CA gets from SC a set of values characterizing the application workload (that is an estimation of the statistical parameters described in 4.2). In our design, the acquired characterization is assumed to be representative of the workload profile of the application for the near-future.

NN is able to predict the average wasted transaction execution time spent by the application, i.e., the average time spent executing aborted transactions, as a function of (A) a given set of values characterizing the workload and (B) a given number of concurrent threads sustaining the application (NN provides an approximation f_N of the function 4.1 and, as we will see in paragraph 5.3, it is able to reach very good prediction performance allowing to accurately determine the wasted time due to transaction aborts). CA exploits NN to calculate, over a range of values for the number of concurrent threads, the expected wasted time that will characterize the application execution in the near future. Then, on the basis of this outcome, CA determines the number of threads that is expected to provide the best application throughput, and keeps active such a number of threads during the subsequent workload sampling interval. A more detailed description of the behaviour of the CA component is given in the next section.

5.1.1 Controlling the Concurrency Level

At the end of each sampling interval, CA gets the vector (rs_s, ws_s, rw_a, ww_a, t_t,ntc_t) from SC. Then, for each k such that $1 \leq k \leq max_{thread}$ (where max_{thread} is the maximum amount of concurrent threads admitted for the application), it generates the vector $\mathbf{v_k} = \{rs_s, ws_s, rw_a, ww_a, t_t, ntc_t, k\}$ and predicts $w_{time,k} = f_N(\mathbf{v_k})$ by relying on NN. After it uses the set of predictions $\{(w_{time,k})\}$ to estimate the number *opt* of concurrent threads which is expected to maximize the application throughput along the subsequent observation period. Specifically, exploiting t_t and ntc_t as predictions of the average execution time of the committed transactions and the *ntc* blocks, respectively, *opt* is equal to the value of k, with $1 \leq k \leq max_{thread}$, for which

$$thr_k = \frac{k}{w_{time,k} + t_t + ntc_t} \tag{5.1}$$

is maximized. Note that $w_{time,k} + t_t + ntc_t$ corresponds to the predicted average execution time between the commit operations of two consecutive transactions along a given thread when there are k active threads. Finally, during the subsequent sampling interval, CA keeps active *opt* threads, deactivating (if active) the remaining $max_{thread} - opt$ threads.

As we have previously stated, $w_{time,k}$ is expressed as a function of t_t and ntc_t. However, these two quantities may depend, in their turn, on the value of k due to different thread contention dynamics on system level resources (e.g. contention on system hardware) when changing the number of threads. As an example, per-thread cache efficiency may change depending on the number of STM threads operating on a given shared-cache level, thus impacting the CPU time required for a specific code block, either transactional or non-transactional. To cope with this issue, we provide correction functions allowing, once known the value of t_t (or ntc_{time}) when running with k threads, which we denote as $t_{t,k}$ and $ntc_{t,k}$ respectively, to predict the corresponding values when supposing a different number of threads. This will lead the final throughput prediction to be actuated via the formula:

$$thr_k = \frac{k}{w_{time,k}(t_{t,k}, ntc_{t_k}) + t_{t,k} + ntc_{t,k}} \tag{5.2}$$

Overall, the finally achieved performance model in Eq. 5.2 has the ability to determine the expected transaction wasted time when also considering contention on system level resources (not only logical resources, namely shared data) while varying the number of threads in the system. In fact, $w_{time,k}(t_{t,k}, ntc_{t_k}) + t_{t,k} + ntc_{t,k}$ corresponds to the predicted average execution time between the commit operations of two consecutive transactions along a same thread when there are k active threads in the system, as expressed by taking into account contention on both logical and system-level resources. So the instantiation of the Neural Network based model for the prediction of $w_{time,k}$ needs to be complemented with a predictor of how t_t and ntc_t are expected to vary vs the degree of parallelism k. Also, the final equation establishing the

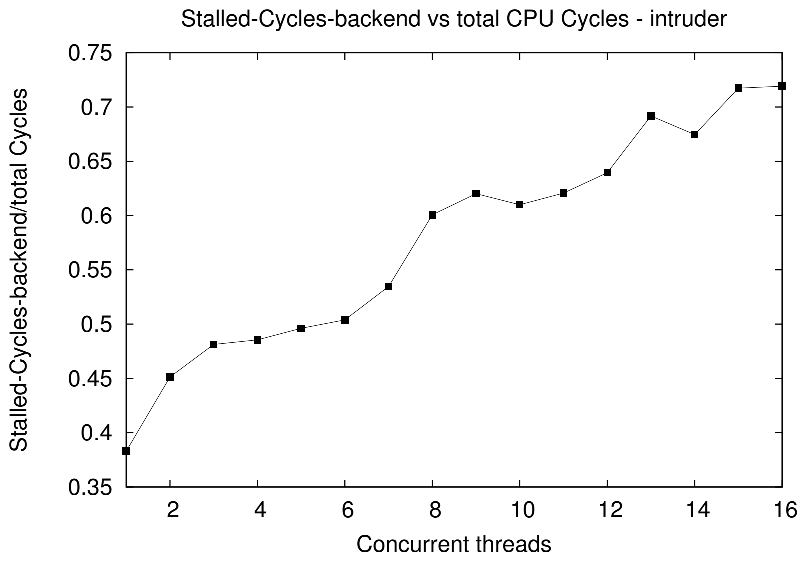


Figure 5.2: Stalled cycles back-end for the intruder benchmark

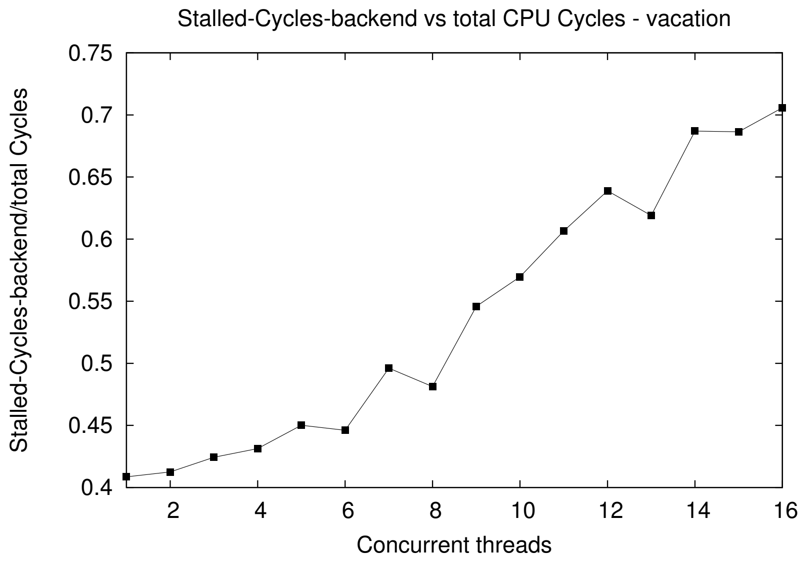


Figure 5.3: Stalled cycles back-end for the vacation benchmark

system throughput, namely Eq. 5.2, which is used for evaluating the optimal concurrency level, also relies on the ability to determine how t_t and ntc_t change when changing the level of parallelism (due to contention on hardware resources). To cope with this issue, we have decided to complement the whole process with the instantiation of correcting functions aimed at determining (predicting) the values $t_{t,k}$ and $ntc_{t,k}$ once know the values of these same parameters when running with parallelism level $i \neq k$. To achieve this goal, the samples used for the training of the Neural Network are used to build, via curve fitting, the function expressing the variation of the number of clock-cycles that the CPU-core spends waiting for data or instructions to come-in from the RAM storage system. The expectation is that the number of clock-cycles spent in waiting phases should scale (almost) linearly vs the number of concurrent threads used for running the application. To support our claim, we report in Figure 5.2 and in Figure 5.3 the variation of the clock-cycles spent while waiting data to come from the RAM storage system for two different STM applications of the STAMP benchmark suite, namely *intruder* and *vacation*, while varying the number of threads running the benchmarks between 1 and 16. (More in detail the graphs shows the ration between the stalled cycles respect to the total cycles varying the concurrency level used to run the application). These data have been gathered on top of the hardware platform already described in section 2.2. The reported statistics have been collected via the `perf` tool, which marks the stall cycles while gathering data from RAM storage as `Stalled-Cycles-Backend`. By the curves the close-to-liner scaling is fairly evident, hence, once determined the scaling curve via regression, which we denote as sc,

$$t_{t,i} = t_{t,k} \times \frac{sc(i)}{sc(k)} \qquad ntc_{t,i} = ntc_{t,k} \times \frac{sc(i)}{sc(k)} \tag{5.3}$$

where:

- $t_{t,i}$ is the estimated expected CPU time (once known/estimated $t_{t,k}$) for a committed transaction in case the application runs with level of concurrency i;
- $ntc_{t,i}$ is the estimated expected CPU time (once known/estimated $ntc_{t,k}$) for a non transactional code block in case the application runs with level of concurrency i;
- $sc(i)$ (resp $sc(k)$) is the value of the correction function for level of concurrency i (resp k).

5.2 Implementation

We have implemented a fully featured Self-Adjusting Concurrency STM (SAC-STM) based on the architecture proposed in the previous sections. The STM layer has been implemented by relying on the release of TinySTM version 1.0 for Unix systems. We used the facilities natively offered by TinySTM to determine w_{time}, t_t and ntc_t. In addition, we instrumented TinySTM code to gather samples to evaluate rs_s and ws_s. In order to compute the access distribution of read/write operations, we added a read counter and a write counter for each element of the lock vector. At the end of the commit phase of a successfully committing transaction, the read (write) counter for each lock associated with an item in the read (write) set of the transaction gets incremented. We make the assumption that the different types of transaction are scheduled with uniform probability between all the threads that execute transactions. This assumption, that will be relaxed in Section 5.4, allows to keep low the overhead associated with the sampling mechanisms. In fact the statistical data gathered by each thread are representative of the whole application workload profile, so we can use only one thread (that we name *master* thread) to collect the statistics used to predict system performance. This choice allows to avoid the usage of synchronization mechanisms inside the added statistic-collection modules and, in turn, it avoids negative side effect on system scalability. Actually, the master thread is randomly selected among the active threads at the beginning of each sampling interval. A sampling interval terminates after the master thread has committed n subsequent transactions. At the end of each sampling interval, the master thread calculates the aggregated statistics and then, by relying on a NN implementation, it calculates the number k of concurrent threads which is expected to maximize the throughput according the approach depicted in Section 5.1.1. Finally, it keeps active k threads (out of the maximum number of max_{thread} threads) during the next sampling interval.

We actually tested two thread activation/deactivation mechanisms. The first one leverages on a shared array with max_{thread} elements. The master thread sets to 1 (0) the elements associated with the threads which have to be deactivated (activated). The slave threads (namely the remaining $max_{thread} - 1$ threads) check their corresponding value before executing a new transaction, by trapping into a busy waiting phase while the value is 1. The second mechanism leverages on a shared array of max_{thread} POSIX semaphores initialized to 0. In this case the master thread increments (decrements) the semaphores associated with the threads which have to be deactivated (activated), and the slave threads, on check, perform a *wait-for-zero* operation on the associated semaphore. Note that in this case, when a thread is deactivated, it actually sleeps (thus not consuming CPU cycles) until it is reactivated. On the other hand, the two different approaches provide different reactiveness since the usage of semaphores imposes sleep-ready thread transitions at the operating system level. In order to further study the effects of thread wake-up and rescheduling, for the case where we rely on semaphores we have studied two different configurations. The first one is such that the k threads to be maintained active (across the max_{thread} threads) are selected according to a round-robin scheme, while the second configuration is such that always the same set of threads are kept active, except for those that have to be newly activated or deactivated, which might reduce the cost of thread reschedule. We note anyway that the latter configuration is not suitable for all kind of applications. Specifically, it can not be used for applications that make a pre-partitioning of the work among threads, because, in this case, sleeping threads may prevent the application to fairly make progress for a relatively long time interval. Overall, the different configurations we consider allow us to study differentiated trade-offs involving both performance and applicability aspects.

Finally, our implementation of NN consists of an acyclic feed-forward full connected network [38] that has been coded by leveraging on FANN open source libraries (version 2.2.0) [67]. NN has an input layer containing seven nodes and an output layer containing a single node, according to the number of input and output parameters of the function f to be estimated.

5.3 Experimental Evaluation

In this section we present the results of an experimental study we carried out to evaluate the effectiveness of our proposal. We run applications from the STAMP benchmark suite on top of the above described implementation of SAC-STM, which has been hosted by the platform described in section 2.2. We present the results for all the STAMP benchmarks. These applications span from low to high percentage of time spent executing transactions (vs non-transactional

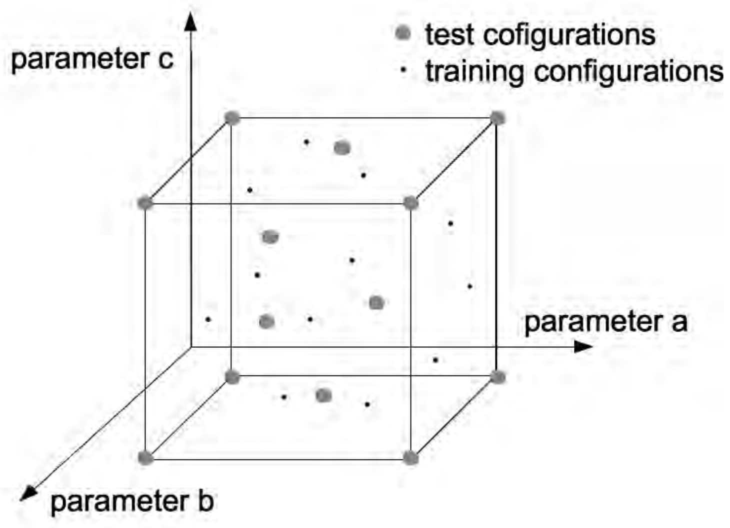


Figure 5.4: Example training configurations and test configurations for the case of three workload configuration parameters.

code blocks), and from low to high data contention levels. When running these applications with different workload configuration parameters on top of the native version of TinySTM the statically configured optimal number of threads can remarkably change or not, depending on the specific benchmark. Hence, the different variability exhibited by these applications in terms of optimality of the number of threads when considering the static case gives rise to a good test-suite for the evaluation of our self-adaptive proposal.

5.3.1 Evaluation Methodology

For each test-bed application we performed an off-line training of NN using samples gathered during the execution of a set of runs of the test-bed application. In order to evaluate the robustness of our proposal, we purposely avoided the generation of training samples that would cover uniformly the sampling space.

Conversely, the training runs have been executed by randomly selecting both the values of the workload configuration parameters of the application, each one in between its corresponding two extreme values depending on the specific parameter, and the number of concurrent threads. Finally, training samples have been randomly selected across samples gathered during the execution of the runs.

We tested SAC-STM using various workload configurations. Further, to assess its robustness, we also used workload configurations associated with the extreme values of the intervals in which the values of the workload configuration parameters have been selected for the NN training phase. An example of the type of workload configurations we used in our tests is depicted in Figure 5.4 for the case of three configuration parameters. The vertices of the cube represent the configurations defined through the extreme values of the intervals. Note that these configurations represent border cases with respect to the configurations used to train NN. Indeed, all the randomly selected configurations used for the training phase are contained within the cube. Since the workload configuration parameters of the applications also affect the number of transactions to be executed, in our tests we excluded those configurations for which the number of transactions within the parallel run was so short not to allow the completion of at least three sampling intervals.

5.3.2 Off-line Training

We trained NN using 800 samples randomly selected over the execution of 64 runs according to the methodology described in the previous section. The number of concurrent threads for each run was randomly selected between 1 and 32

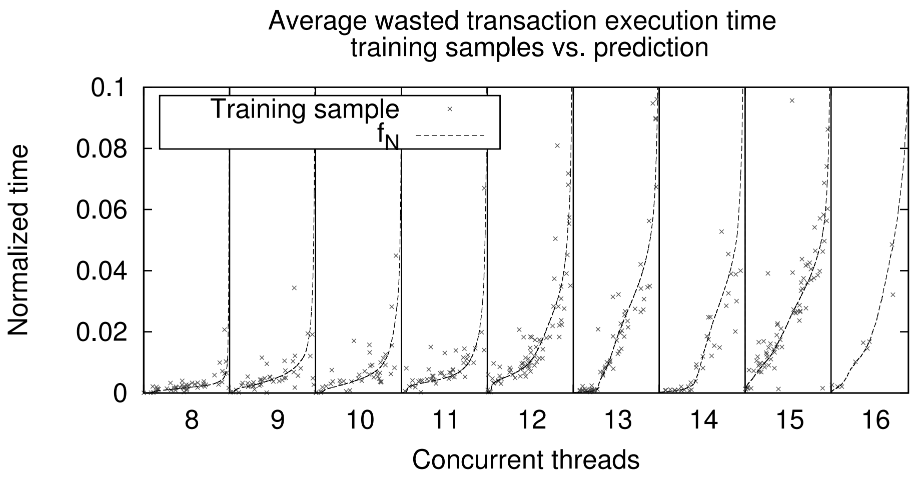


Figure 5.5: Average wasted transaction execution time: training set vs. predicted.

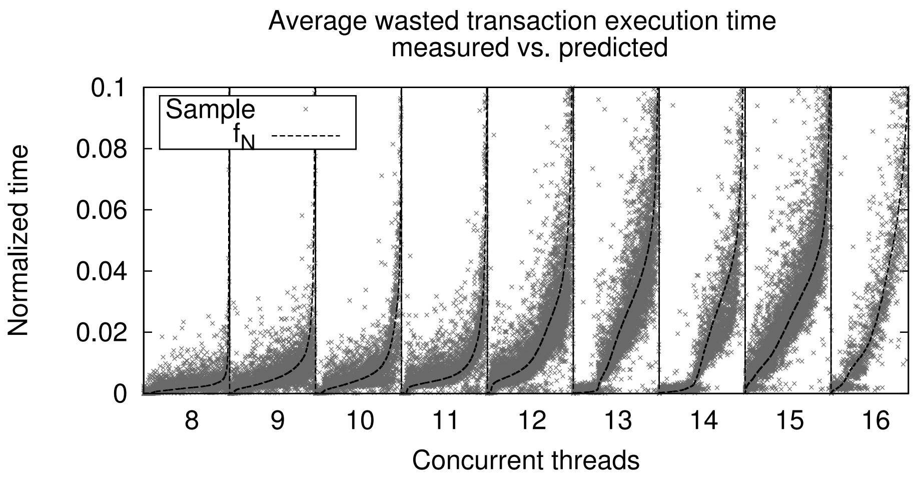


Figure 5.6: Average wasted transaction execution time: measured vs. predicted.

([1]). Each sample contains the values calculated over a sampling interval whose duration was determined by the execution of 4000 subsequent committed transactions along the same thread. To limit the number of outliers, we discarded (filtered out) samples stemming from sampling intervals in which more than the 99% of the transaction runs have been aborted. Note that, when moving towards such an abort probability, the transaction response time grows very fast and exhibits high variability. We assume that in this situation the system throughput is never optimal. Hence, for higher abort probability it suffices that NN approximates the transaction response times by relying on the closest samples which have not been filtered out.

To train NN we used a back-propagation algorithm [68, 69, 70]. We observed that a number of hidden layers equal to one was a good thread-off between prediction accuracy and learning time. In this case, the number of hidden nodes for which NN provided the best approximations was between 4 and 16, depending on the application. Further, during our experiments, we observed that values between 0.0 and 0.2 were good for both the learning coefficient and the momentum, respectively. In the worst cases, the iterations of the back-propagation algorithm have been no more than 2500, and the algorithm execution time was less than 10 seconds on a desktop machine equipped with an Intel®Core^{TM}2 Duo P8700 and 8 GB RAM. On the other hand, the on-line computation by NN was on the order of a few microseconds.

To provide some graphical details about off-line training, in Figure 5.5 we plotted the dispersion of the training set values of the (normalized) wasted trans-

[1]Although it is generally not convenient to use more threads than the available cores, for completeness of the analysis (and for compliance with what done in other studies), we also report some performance data related to the case where max_{thread} is varied up to 32, thus doubling the 16 CPU-cores available on the used hardware platform. This is the reason why we considered up to 32 threads in the off-line learning phase.

action execution time, namely w^t_{time}, together with the function f_N learnt by NN as result of the off-line training. These data refer to the intruder benchmark. The plots refer to different values of the number of concurrent threads, that has been varied between 8 and 16. Note that, fixed the number of concurrent threads, the values of f_N depend on other six parameters which have been projected on a two-dimensional space, where the f_N function is plotted according to an increasing ordering of its values. In order to assess the quality of the prediction by NN, in Figure 5.6 we plotted the estimated function f_N and the wasted transaction execution time associated with a larger set of training samples that have not been used for the aforementioned training phase. As we can note, also in cases where very few training samples are used (see, e.g., Figure 5.5 in correspondence to 16 concurrent threads), NN is able to reliably predict the wasted transaction execution times (see Figure 5.6 in correspondence to the same number of concurrent threads) thanks to its interpolation/extrapolation ability.

5.3.3 Results

For all the tests we present in this section, we plot the application execution time (expressed in sec) achieved with SAC-STM and with the original TinySTM, which we use as a baseline, while varying max_{thread}. We initially consider test cases where max_{thread} is less than or equal to the value 16, which corresponds to the amount of CPU-cores available on the used hardware platform. After (as already hinted, for completeness of the analysis) we present the results for a test where we consider values for max_{thread} up to 32. When considering values of max_{thread} less than or equal to 16, the selected mechanism for managing thread activation/deactivation within SAC-STM is busy-waiting, which for the specific configuration conditions does not give rise to delays in the progress of individual threads (since each thread runs on an exclusively dedicated CPU-core) and avoids sleep-ready thread transitions at the operating system level. For each test-bed application, we present the results achieved with four different workload configurations. These include configurations corresponding to some vertices of the cube (see Section 5.3.1) where SAC-STM achieved the worst and the best performance with respect to the best case observed when running on top of TinySTM while manually varying the number of threads.

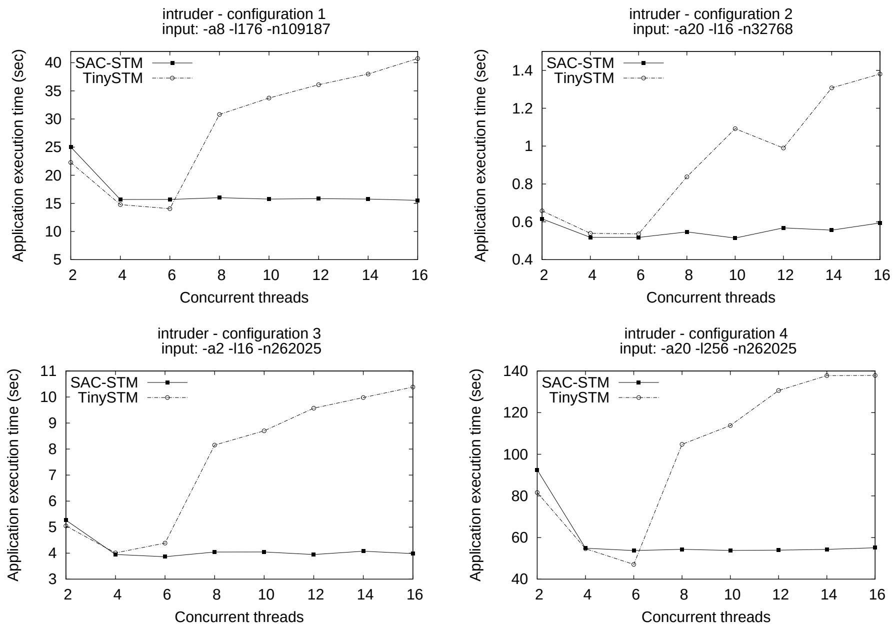


Figure 5.7: Application execution time - intruder

In Figure 5.7 we present the results for the intruder benchmark. As we can see, for all considered configurations, the application execution time achieved with TinySTM decreases when increasing the number of used threads up to 4-6, while for greater values it drastically increases. Conversely, for all the tests, SAC-STM achieves very good results independently of the maximum amount of allowed concurrent threads. In fact, in scenarios where max_{thread} is less than the amount of threads giving rise to the optimum case for TinySTM, the results achieved with SAC-STM and TinySTM are comparable. With more threads, SAC-STM is able to constantly ensure an application execution time very close to the best one achieved with TinySTM. In particular, also for the two configurations corresponding to the vertices of the cube, i.e., *configuration 3* and *configuration 4*, the performance results are good. In the most adverse case to SAC-STM, which corresponds to *configuration 4*, SAC-STM constantly achieves an execution time of no more than 12.5% worse compared to the best case provided by TinySTM, i.e. when it runs with a fixed number of 6 threads. However we note that as soon as 8 or more threads are used by TinySTM, its performance rapidly degrades up to a factor 2.8x, thus exhibiting a clear scalability problem with this workload. This phenomenon is avoided at all by SAC-STM thanks to its proper thread activation/deactivation functionalities, which provide a means to control the negative effects associated with data contention.

The results of the genome benchmark tests are shown in Figure 5.8. For *configuration 1* and *configuration 4* (the latter is a vertex configuration of the cube), the execution times with SAC-STM and TinySTM are comparable up to 4 threads. With more threads, while the performance with TinySTM degrades, SAC-STM ensures, again independently of the number of available threads, an execution time comparable or lower than the best one provided by TinySTM (i.e. with 4 threads). With *configuration 3* (which is the other vertex configuration of the cube) the best execution time with TinySTM is achieved with 14 threads, after which the performance slightly decreases. With this configuration, the results achieved with SAC-STM are, on average, comparable with those by TinySTM. In the most adverse case to SAC-STM, which corresponds

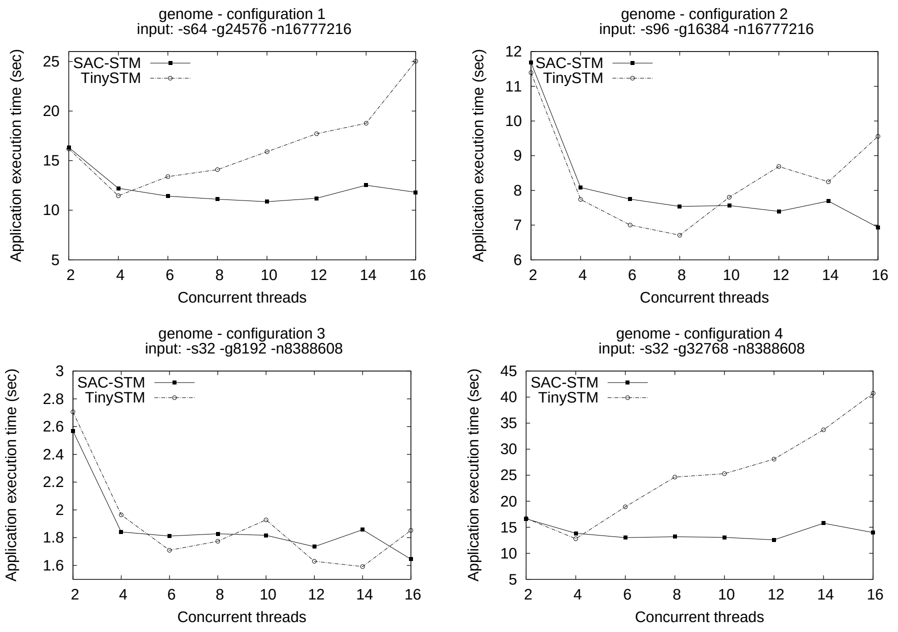


Figure 5.8: Application execution time - genome

to *configuration 2*, SAC-STM constantly achieves an execution time of no more than 11% worse compared to the best case provided by TinySTM, i.e. when it runs with a fixed number of 8 threads.

The results plotted in Figure 5.9 refer to the tests with the kmeans benchmark. Also in this case, SAC-STM provides performance benefits in all scenarios when max_{thread} is set to a value larger than the value giving rise to the best case for TinySTM. For *configuration 4*, namely a vertex configuration of the cube, the execution times are comparable while varying the amount of available threads. For the other vertex configuration, namely *configuration 3*, the best execution time achieved by TinySTM (i.e., with 4 threads) is about 22% lower than the execution time achieved with SAC-STM. But with more available threads, SAC-STM constantly achieves a better execution time, hence outlining again how TinySTM may suffer from selection of an oversized degree of concurrency,

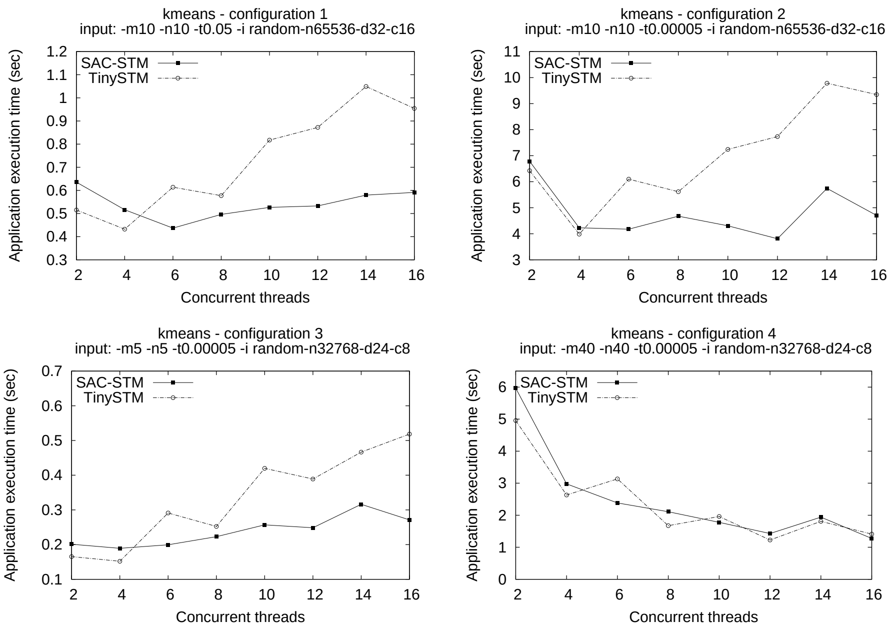


Figure 5.9: Application execution time - kmeans

which is instead not the case for SAC-STM.

In Figure 5.10 we can see the results related to the tests with labyrinth benchmark. SAC-STM provides performance benefits in all the scenarios where a number of thread greater than the one that allows to obtain optimal performance with Tiny-STM is used. In *configuration 1*, *configuration 3* and *configuration 4* we can see how the performance rapidly decreases using more than 6 thread with TinySTM. With SAC-STM instead the performace remains near the optimum for each level of parallelism greater than 6. For *configuration 2*, we can see that the number of available cores doesn't allow to reach the concurrency level that produce the amount of logical contention necessary to penalize the performance. In this case we can see that SAC-STM allows to obtain the same performance of TinySTM.

The Figure 5.11 shows test results for ssca2 benchmark. In all the tests we

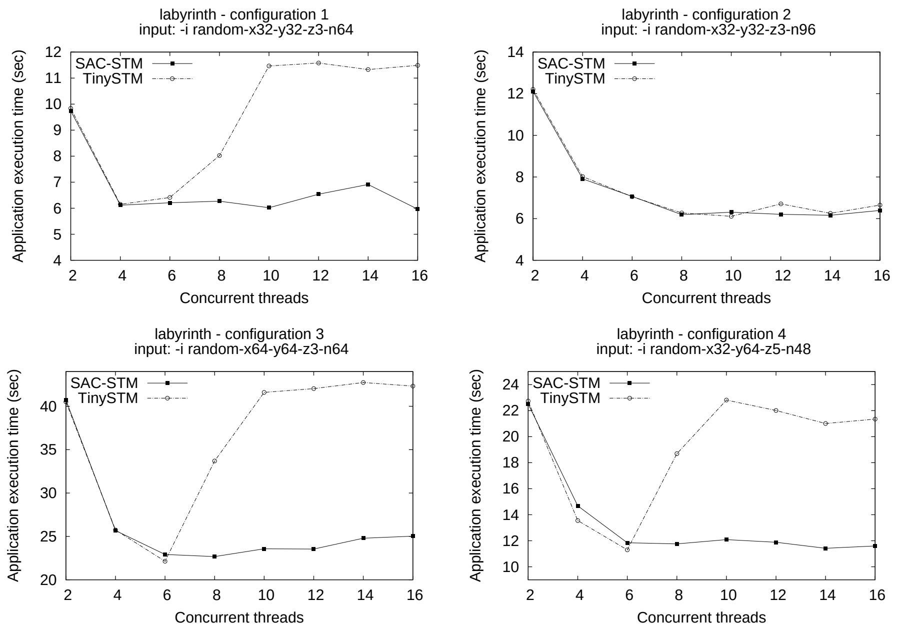


Figure 5.10: Application execution time - labyrinth

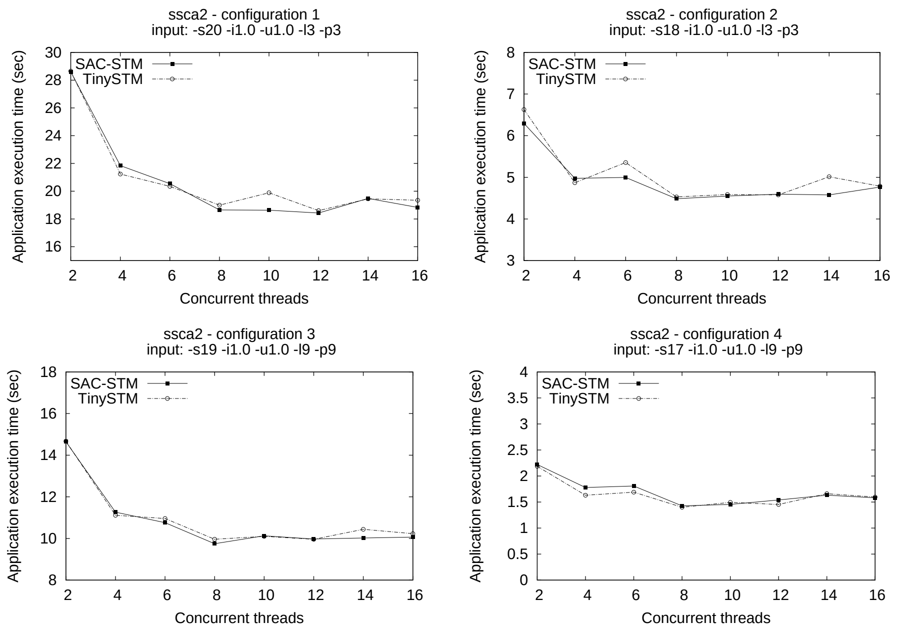


Figure 5.11: Application execution time - ssca2

made with this benchmark we verified that the number of available cores is not sufficient to reach the concurrency level that produce performance degradation. As we can see from the graphs with this benchmark SAC-STM allows always to obtain the same performance of TinySTM.

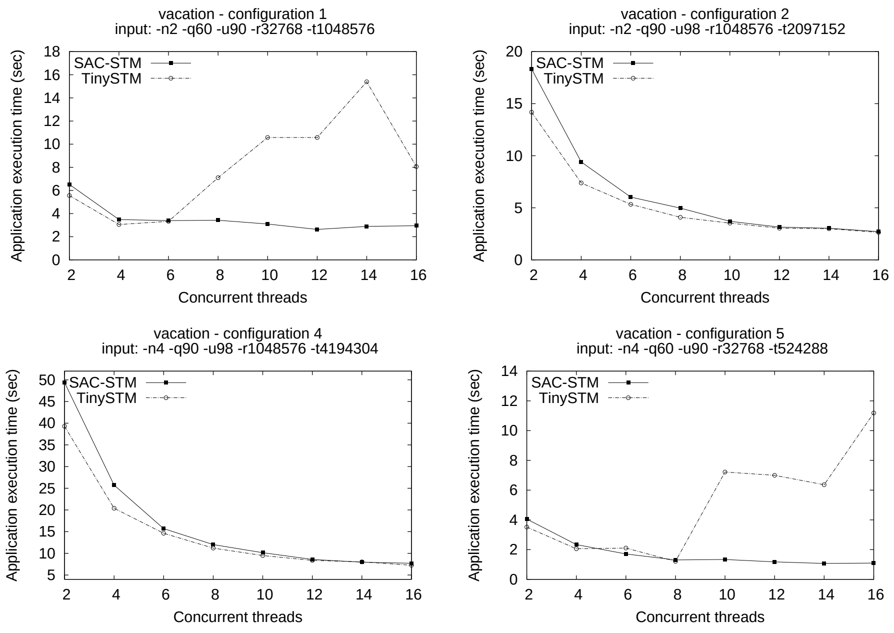


Figure 5.12: Application execution time - vacation

The test results of vacation benchmark are showed in Figure 5.12. From *configuration 1*, *configuration 4* we can see that this application reaches optimal performance with different levels of parallelism, depending on the specific workload. More precisely we can see that in *configuration 1* the optimal level of parallelism with TinySTM is around 4 and in *configuration 4* it is around 8. From the graphs we can see how SAC-STM ensure the same performance of TinySTM until it reach the optimum. Beyond this point the performance of TinySTM degrades, conversely SAC-STM allows to obtain always performance near to the optimum. In *configuration 2* and *configuration 3* the graphs show that the number of available cores doesn't allow to reach the level of parallelism that give the optimal performance, because in that scenarios the logical contention on shared data is very low. In this cases too, SAC-STM allows to obtain the same performance levels reached by TinySTM.

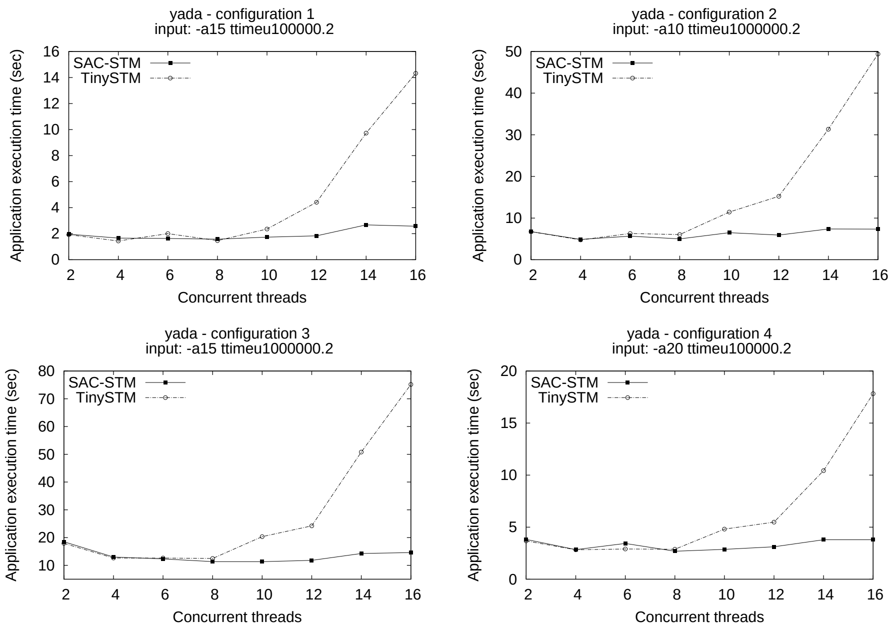


Figure 5.13: Application execution - yada

In Figure 5.13 we can see the test results for yada benchmark. All the examined scenarios give similar results. As we can see from the four graph the application reaches the optimum using a number of thread between 6 and 8. Behind this threshold the performance obtained with TinySTM degrades. Instead SAC-STM still allows to obtain performance near the optimum independently from the maximum level of parallelism admitted.

The last examined benchmark, that is the last in the STAMP benchmark suite, too, is bayes. The test results are showed in Figure 5.14. As we can see from the graphs the optimal performance is reached by TinySTM with a

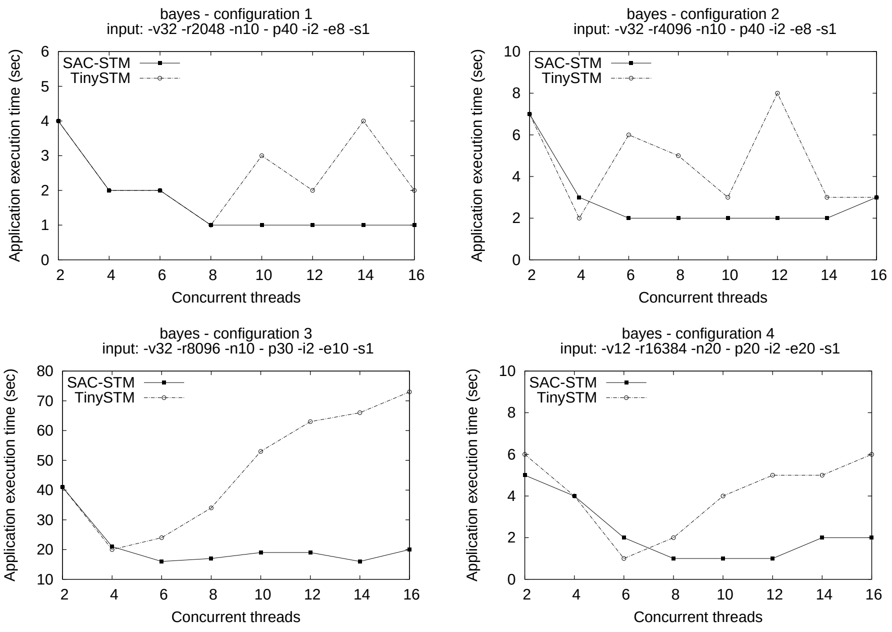


Figure 5.14: Application execution - bayes

level of parallelism between 4 and 8 threads, depending on the benchmark input parameters. The graphs show that, increasing the maximum available level of parallelism, SAC-STM ensures the same performance of TinySTM until the optimum is reached. Behind this point the TinySTM performance degrades while SAC-STM continues to provide performance near the optimum.

In order to show the different effects on performance determined by the specific thread activation/deactivation mechanisms, in Figure 5.15 we plotted the application execution time obtained with SAC-STM in a test with the intruder benchmark, where we used busy-waiting or semaphores. For the latter case, we also report data related to both round-robin and non-round robin policies for the selection of the threads to be kept active during the subsequent observation period. Additionally, we plotted the application execution time obtained with TinySTM. This time we considered values of max_{thread} up to 32. We note that the estimated optimal number of threads by the NN in this test was always around 5. We can see by the plots that differences between the execution times are low when considering up to 6 threads. After, the execution time with TinySTM quickly increases. With busy-waiting the execution time by SAC-STM remains optimal up to 16 threads, then it quickly increases, as expected by the fact that only 16 CPU-cores are available (thus leading busy-waiting threads to interfere on the advancement of the threads actively supporting the application). With semaphores, the execution time increases between 6 and 10 threads. Then it tends to remain quite constant, even after 16 threads. This is due to the fact that, up to 5 threads, none or a few threads, for each sampling phase, get context-switched due to the round-robin selection, giving way to other threads which are resumed. With more than 5 threads, and up to 10 threads, the number of threads getting context-switched progressively increases, causing an increase of the execution time. With more than 10 threads, 5 threads, on average, get context-switched and 5 threads are resumed for each sampling period, then the cost of these operations tend to remain constant when further increasing the number of threads of the application. Finally, using semaphores without round-robin selection, the execution time remains quite close to the optimum case independently of the number of threads. In fact, avoiding round-robin selection, running threads tend to remain the same, so reducing costs associated with wait-ready transitions (namely sleeping-thread resuming) and context-switching.

5.4 Sampling Overhead Optimization

Until now we did the assumption that the different types of transaction are scheduled with uniform probability between threads. This assumption allows

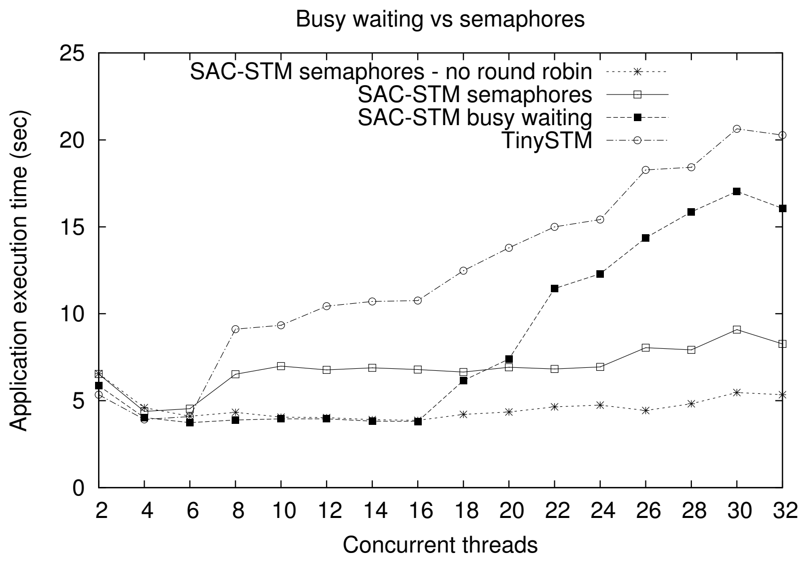


Figure 5.15: Application execution time with intruder up to 32 threads.

to consider the samples taken by any thread like representative of the whole application workload. In this scenario it is possible to use only one thread for sampling collection during the application execution without loss of accuracy. Using only one thread for this task allows to reduce the sampling overhead that can negatively affect the whole application performance. In fact, as showed by the curves in the graphs from 5.7 to 5.14 that compare Tiny-STM with SAC-STM, the sampling overhead can be considered negligible. However, if we relax the assumption of uniform distribution of the workload, we can't consider the samples taken by any thread like representative of the whole application workload. In this case each thread must collect its own statistics that must be merged to obtain a sample representative of the whole application workload. This last procedure produces more overhead than the one that use only one thread as we will show on paragraph 5.5.2. So, to avoid a strong performance degradation, the sampling collection activity must be redesigned with the aim of reduce as soon as possible the sampling overhead. To do so, three main aspects must be taken into account:

- if we use only one worker thread (i.e. thread that execute transactions) to collect input parameters data, the overall input parameters collection overhead decrease increasing the level of parallelism.
- the sampling overhead is inversely proportional to the sampling period, that corresponds to the number of committed transaction taken into account to build a sample;
- not all the parameters used for transactional workload characterization have the same sampling costs. As we will show in paragraph 5.5.2 for the parameters rs_s, ws_s, t_t, ntc_t the costs can be very low but for rw_a and ww_a the costs can be high.

We relaxed the assumption of the uniform distribution of the workload so the possibility of use only one thread for sampling collection must be discarded. About the sampling period, we can see that it not only affect the overhead, but the samples accuracy, too. To obtain samples that properly describe the workload, the sampling period must be carefully tuned: if we choose a excessively short period the sample could be not statistically significant, otherwise if we choose a excessively long period there is the risk that the samples are not able to describe some fast workload variations. To preserve the prediction accuracy, we decided to choose the size of the sampling period so that it can ensure the highest precision in workload description, without taking into account the effects on sampling overhead. The reason for this choice is that a wrong decision about the optimal concurrency level has a greater negative impact on performance respect to the once due to the sampling overhead. So we have only one way to reduce the sampling overhead: we have to use as few as possible input parameters for performance prediction. To do so, we have to analyse the collected data to understand if the monitored parameters are all strictly necessary to perform accurate predictions. During the analysis process we have to take into account that a general application can be characterized by:

- a statical workload: that is the behaviour of the parameters used to characterize the application workload doesn't vary during the application lifetime;
- a dynamic workload: that is the behaviour of the parameters used to characterize the application workload can vary during the application lifetime.

In the first case the minimum set of parameters *minSet* necessary to obtain an accurate performance prediction can be established just one time and than it will be valid for all the application lifetime. In the second case *minSet* can vary from 1 to the maximum number of collectable parameters, so in this case it is necessary a dynamic mechanism to choose the content of *minSet*. The statical workload case can be considered a special case of the dynamic workload one, so in the next paragraphs we will propose a solution that dynamically change the size of *minSet* with the aim of reduce the sampling overhead preserving at the same time the accuracy of the performance predictor.

5.5 Dynamic feature selection

5.5.1 Rationale

Our rationale for the definition of an innovative approach where, depending on the current execution profile of the application, the set of input features to be sampled can be dynamically shrunk, or enlarged towards the maximum cardinality, is based on noting that:

A: some feature values may show small variance during a given time window, and/or

B: some feature values may be statistically correlated (also including the case of negative correlation) to the values of other features during a given time window.

Particularly, we can expect that (significant) variations of w_{time}, if any, do not depend on any feature exhibiting small variance over the current observation window. On the other hand, in case of existence of correlation across a (sub)set of different features, the impact of variations of the values of these features on w_{time} can be expected to be reliably assessed by observing the variation of any individual feature in that (sub)set. In case the above scenarios occur, we note it can be possible to build an estimating function for w_{time} which, compared to f, relies on a reduced number of input parameters. Consequently, NN has to estimate a simpler f_N function. On the other hand, the relevance of excluding specific input features lies on the potential for largely reducing the run-time overhead associated with application sampling, as we shall demonstrate later on in the chapter.

For the reference set $\{rs_s, ws_s, rw_a, ww_a, t_t, ntc_t\}$, it comes out natural to think about the following expectations in relation to the correlation of subsets of the input features:

- the size of the transaction read-set/write-set may be correlated to the transaction execution time. In fact, the number of read (write) operations executed by the transaction directly contributes to the actual transaction execution time. If this reveals true, t_t and one feature, selected between rs_s and ws_s, can be excluded;

- read-write and write-write conflict affinities may exhibit correlation. In fact, these two indexes are both affected by the distribution of the write operations executed by the transactions. If this reveals true, rw_a or ww_a can be excluded.

Overall, we have some expectations for the actual occurrence of the condition expressed by point **B** for at least a subset of the input features. Further, depending on the actual application logic, and the associated execution profile, generic sets of features could result correlated over a given time window, even in case they are not directly affected by each other, as instead it occurs for, e.g., the couple formed by rs_s and t_t. Additionally, still depending on the application logic, any of the features in the set $\{rs_s, ws_s, rw_a, ww_a, t_t, ntc_t\}$ may exhibit small variance over a given time window, thus being candidate to be excluded from the relevant set of input features.

To determine at what extent such an expectation materializes, and to observe whether the scenarios in points **A** and **B** can anyhow materialize independently of the initial expectation, we have performed an experimental study relying on the complete suite of STM applications specified by the STAMP benchmark. Particularly, we report in Table 5.1 data related to the observed correlation among the different features. All data refer to serial executions of the STAMP benchmark suite, which have been carried out on the hardware platform already described in section 2.2.

We note that serial execution is adequate for the purpose of this specific experimentation since it is only tailored to determine workload features that are essentially independent of the degree of parallelism in the execution. Specifically, given that correlation and variance are computed over feature-samples, each one representing an average value (over a set of individual samples taken along one observation window entailing 4000 transactions in this experiment), for the only parameters that can be potentially affected by hardware contention (e.g. bus-contention) in case of real parallelization, namely t_t and ntc_t, the corresponding spikes (if any) would be made relatively irrelevant by the aggregation of the individual samples within the window related average.

The data confirm that the rationale behind our proposal can find justification in the actual behavior of STM applications, when considering the execution patterns provided by the STAMP suite as a reliable representation of typical STM applications' dynamics. In fact, by Table 5.1, we can observe that the correlation between rw_a and ww_a is higher than 0.8 for 4 applications (out of the 8 belonging to the STAMP suite), the correlation between rs_s and t_t is higher than 0.8 for 3 (out of 8) applications, and the correlation between ws_s and t_t is higher than 0.8 for 2 (out of 8) applications. Further, as reported in Table 5.2, we observed very reduced variance for rw_a and/or ww_a for many of the applications, and reduced variance for rs_s and/or ws_s in a few cases.

5.5.2 Pragmatic Relevance: Run-time Sampling Costs

The pragmatic relevance of an approach where the sampled input features to be provided to the NN gets shrunk to a minimal set, which is anyhow sufficient to capture the workload characteristics, is clearly related to the possibility to reduce the sampling overhead. This is a very relevant issue to cope with, given that sampling needs to be carried out along the critical path of application processing. Specifically, it needs to capture run-time dynamics related to the same threads that are in charge of executing transaction (such as evaluating the

ssca2						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,259	1		-	-	-
rs_s	-0,166	0,190	1	-	-	-
ws_s	-0,166	0,190	1	1	-	-
rw_a	-0,024	-0,638	-0,136	-0,136	1	-
ww_a	-0,001	-0,629	-0,210	-0,210	0,992	1

yada						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,705	1	-	-	-	-
rs_s	0,860	0,619	1	-	-	-
ws_s	0,828	0,617	0,946	1	-	-
rw_a	-0,417	-0,183	-0,508	-0,552	1	-
ww_a	-0,400	-0,173	-0,491	-0,542	0,999	1

intruder						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,781	1	-	-	-	-
rs_s	0,914	0,940	1	-	-	-
ws_s	0,577	0,924	0,848	1	-	-
rw_a	0,516	-0,377	-0,540	-0,330	1	-
ww_a	0,023	-0,350	-0,269	-0,559	0,322	1

vacation						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,989	1	-	-	-	-
rs_s	0,507	0,520	1	-	-	-
ws_s	0,345	0,315	-0,487	1	-	-
rw_a	-0,167	-0,179	0,811	0,657	1	-
ww_a	-0,572	-0,535	0,262	-0,954	-0,483	1

genome						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,012	1	-	-	-	-
rs_s	0,352	0,902	1	-	-	-
ws_s	-0,742	0,492	0,158	1	-	-
rw_a	-0,584	-0,202	-0,397	-0,422	1	-
ww_a	0,040	-0,027	-0,009	-0,049	0,064	1

labyrinth						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,993	1	-	-	-	-
rs_s	0,992	0,991	1	-	-	-
ws_s	0,992	0,992	0,999	1	-	-
rw_a	-0,521	-0,500	-0,495	-0,492	1	-
ww_a	-0,332	-0,277	-0,273	-0,267	0,714	1

kmeans						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,141	1	-	-	-	-
rs_s	0,434	0,194	1	-	-	-
ws_s	-0,524	0,106	0,481	1	-	-
rw_a	-0,245	-0,729	0,072	0,177	1	-
ww_a	-0,072	-0,723	0,090	0,008	0,968	1

bayes						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,141	1	-	-	-	-
rs_s	0,434	0,194	1	-	-	-
ws_s	-0,524	0,106	0,481	1	-	-
rw_a	-0,245	-0,729	0,072	0,177	1	-
ww_a	-0,072	-0,723	0,090	0,007	0,968	1

Table 5.1: Input features correlation for the applications with the STAMP benchmark suite

	ssca2	**intruder**	**genome**	**kmeans**
t_t	$1,14 \cdot 10^5$	$1,24 \cdot 10^7$	$6,05 \cdot 10^7$	$5,07 \cdot 10^5$
ntc_t	$2,27 \cdot 10^4$	$1,19 \cdot 10^7$	$1,09 \cdot 10^6$	$1,65 \cdot 10^6$
rs_s	$1,5 \cdot 10^{-3}$	142	945	0,134
ws_s	$1,3 \cdot 10^{-3}$	4,311	0,958	7,61
rw_a	$2,87 \cdot 10^{-10}$	$1,25 \cdot 10^{-5}$	$7,46 \cdot 10^{-10}$	$3,34 \cdot 10^{-4}$
ww_a	$1,03 \cdot 10^{-10}$	$1,17 \cdot 10^{-3}$	$4,95 \cdot 10^{-4}$	$4,16 \cdot 10^{-4}$
	yada	**vacation**	**labyrinth**	**bayes**
t_t	$7,01 \cdot 10^6$	$8,82 \cdot 10^6$	$3,24 \cdot 10^{12}$	$5,07 \cdot 10^5$
ntc_t	$7,38 \cdot 10^4$	$2,57 \cdot 10^5$	$1,04 \cdot 10^7$	$1,65 \cdot 10^6$
rs_s	33	770	70	0,134
ws_s	1,914	7,5	173	7,614
rw_a	$4,60 \cdot 10^{-6}$	$4,16 \cdot 10^{-13}$	$4,02 \cdot 10^{-7}$	$3,34 \cdot 10^{-4}$
ww_a	$2,44 \cdot 10^{-5}$	$1,76 \cdot 10^{-10}$	$2,04 \cdot 10^{-7}$	$4,16 \cdot 10^{-4}$

Table 5.2: Input features variance for the applications with the STAMP benchmark suite

transaction duration by taking a snapshot of the system real-time-clock right upon starting up, or ending, the execution of the transaction along the thread).

An alternative approach to reducing such an overhead would be to make an individual thread (over a set of m concurrent threads) to take samples and to provide statistical data while the application is running. This would delay the critical path execution of 1 out of m threads. However, this approach exhibits two drawbacks that make it unsuited for generic settings:

- The production frequency for the samples gets reduced. Hence, catching any variation in the execution profile of the application may occur untimely.

- STM applications may devote specific threads to run specific transactions (e.g., for locality along the thread execution and cache efficiency improvement [71]). Hence, taking samples along a single thread does not provide a complete picture of the application workload, even in case the sampling thread is dynamically changed over time (e.g. in round-robin fashion).

Further, significant overhead reduction could be achieved via the above approach only for values of m which are larger than the optimal degree of parallelism for the specific application. This may lend the approach not to be viable for specific system setting biased to the optimization of the concurrency level and reduction of the transaction wasted time.

Overall, the typical scenario for reliable sampling and workload characterization (and timely determination of shifts in the workload behavior) consists of taking samples for evaluating the input features to be provided to NN along the execution of all the active concurrent threads.

For this reference scenario, we have experimentally evaluated the sampling overhead for STAMP applications while varying (a) the number of concurrent threads (between 1 and 16), and (b) the set of selected input features. The overhead value has been computed as the percentage of additional time required to complete the execution of the benchmark application in case sampling is activated, compared to the time required for executing the application in case sampling is not activated. The platform used for the experiments is described in section 2.2. From Figure 5.16 to Figure 5.23 we report the overhead values for all the application in the STAMP benchmark suite. For completeness, we include graphs that shows the overhead amount in terms of both relative and absolute values.

One important observation we can make when analyzing the results is that, once fixed the set of input features for which sampling is active, the overhead tends to scale down while the number of concurrent threads gets increased. This behaviour is related to a kind of throttling effect manifesting within the system in case any active thread is involved in the sampling process (since

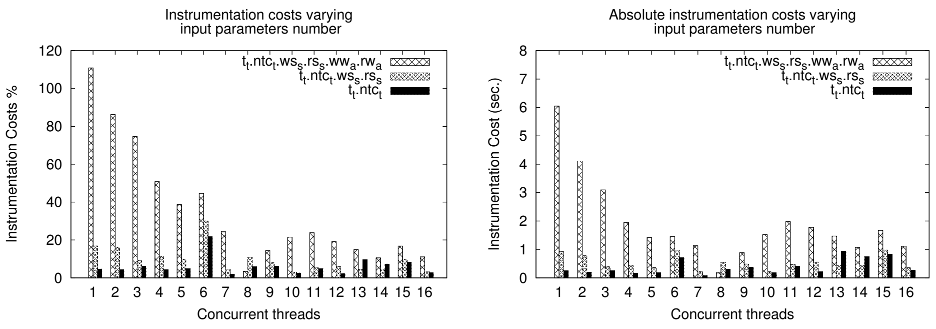


Figure 5.16: Relative and absolute instrumentation costs - intruder

threads issue transactions with reduced rate, just due to delays associated with sampling activities along thread execution). Particularly, when the degree of concurrency gets increased the throttling effects tends to reduce the number of transaction aborts, which tends to reduce the overhead observed when running with the sampling process active, with respect to the case where sampling is not activated. This is a typical behavior for parallel computing applications entailing optimistic processing and rollback actions [72] (just as it occurs for transactions in STM systems). This phenomenon is not observed for reduced values of the number of threads, which leads to reduced contention levels, and hence to reduced abort probability, for which non-significative positive effects can be achieved by to throttled execution.

Another observation we can make when analyzing the results is that, when considering the case of the maximum set of sampled input features, the overhead tends to scale down while the number of concurrent threads gets increased. However, the most significant reduction of the overhead is observed exactly for the cases where the set of input features for which sampling is active gets shrunk. For example, as showed in the first graph of figure 5.16 for the intruder benchmark, when shrinking the monitored features from 6 to 4 or 2, we get up to 90%

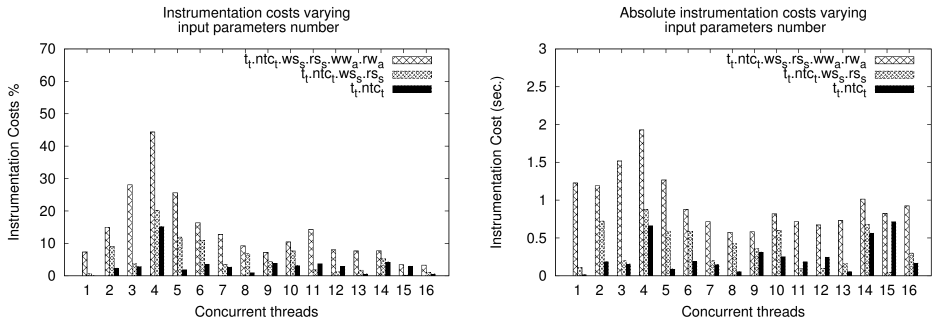


Figure 5.17: Relative and absolute instrumentation costs - bayes

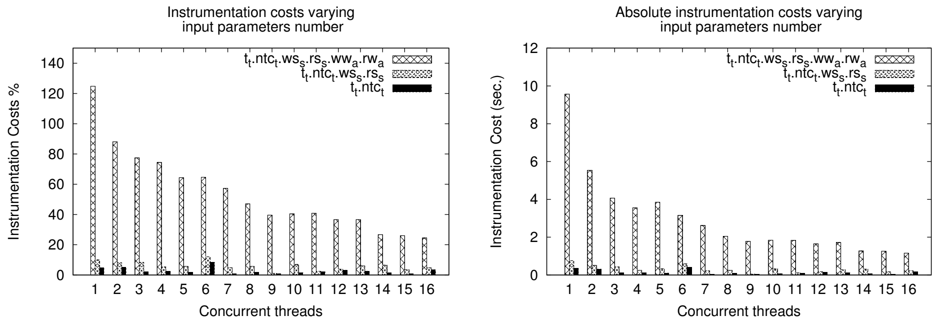


Figure 5.18: Relative and absolute instrumentation costs - ssca2

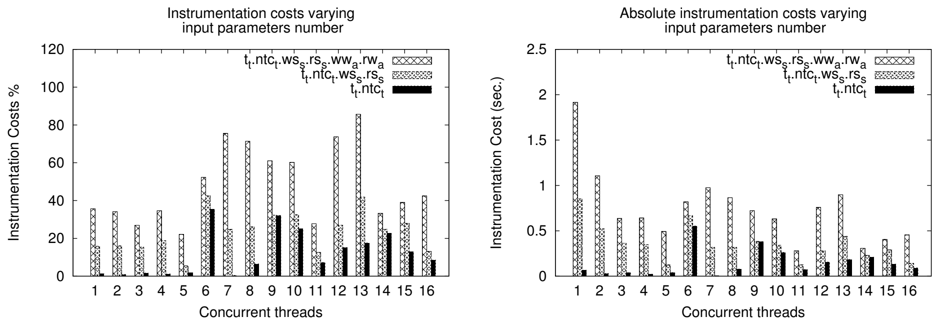


Figure 5.19: Relative and absolute instrumentation costs - vacation

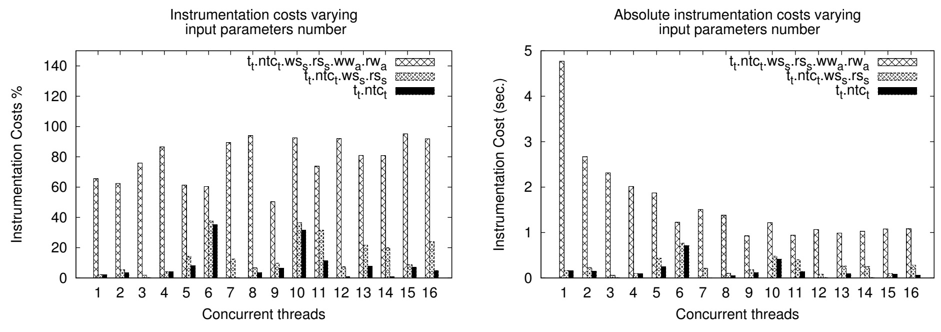


Figure 5.20: Relative and absolute instrumentation costs - kmeans

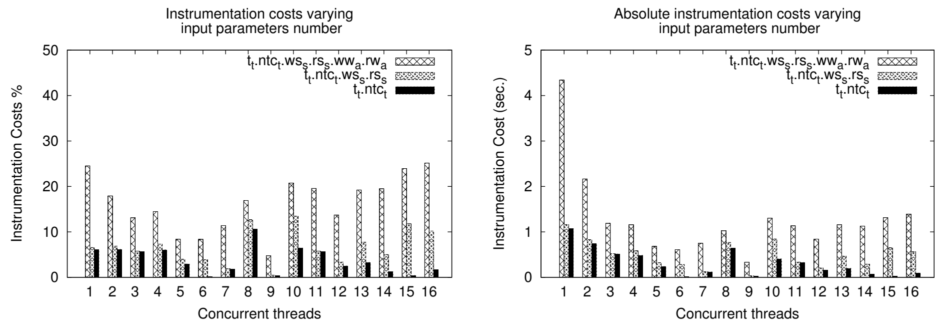


Figure 5.21: Relative and absolute instrumentation costs - labyrinth

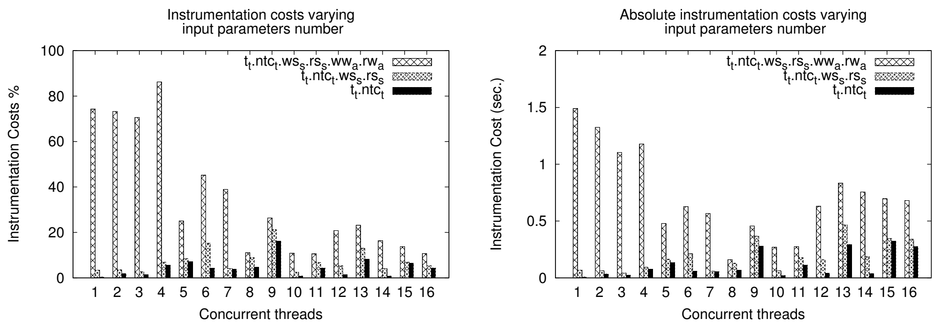


Figure 5.22: Relative and absolute instrumentation costs - yada

reduction of the overhead for smaller lower values of the number of concurrent threads (namely up to 6). This is highly significative when considering that the optimal degree of parallelism for intruder has been shown to be around 5-6 when running on the same hardware platform used in this study (see section 5.3.3). In other words, the optimal parallelism degree is achieved for a number of concurrent threads that does not allow affording the overhead due to sampling in case no optimized scheme for shrinking the set of features to be sampled is provided.

5.5.3 Shrinking vs Enlarging the Feature Set

As pointed out in previous paragraphs, shrinking the set of features to be provided in input to the neural network based performance model can rely on runtime analysis of variance and correlation. However, the application execution profile may vary over time such in a way that excluded features become again relevant. As an example, two generic features x and y, which exhibited correlation in the past, may successively start to behave in an uncorrelated manner. Hence, the excluded feature (x or y), if any, should be re-included within the input set, since both of them are again relevant for reliably characterizing the workload.

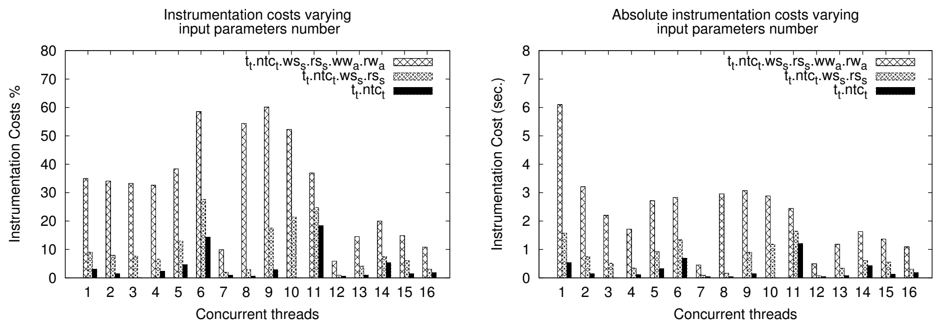


Figure 5.23: Relative and absolute instrumentation costs - genome

Detecting this type of scenarios, in order to support the dynamic enlarging of the feature-set cannot be based on run-time input features analysis (e.g. analysis of the correlation), since the feature that was excluded from the input set (for overhead reductio purposes) has been no more sampled. Hence, no fresh information for that feature is available to detect whether variance and/or correlation with other features have changed.

To overcome this problem, our proposal relies on evaluating whether the current wasted-time prediction by NN is of good quality or not (compared to the real one observed at run-time during the successive observation window). In case the quality is detected to be low, the input feature set can be enlarged towards the maximum in order to recover to a good workload characterization scenario. In other words, low quality prediction by NN is imputable in our approach to the reliance on an input feature-set currently expressing a wrong/not-complete characterization of the workload.

The actual index we have selected for determining the quality of the prediction is the *weighted root mean square error* (WRMS) of the NN wasted time prediction vs the corresponding real value measured in the system. To provide quantitative data showing that WRMS can be considered as a reliable metric, we have performed additional experiments where the effects of concurrency regulation performed by the original version of SAC-STM have been compared with the observed values of WRMS. This experimentation has been carried out by varying both the number of hidden nodes within the NN used by SAC-STM and the number of iterations of the used NN training algorithm. Variations of these parameters allowed us to generate differentiated configurations where the NN may exhibit differentiated prediction qualities.

The results by this experimentation are reported in Figures 5.24 and 5.25, respectively for the case of the vacation and intruder benchmark applications. However the data obtained with the remaining stamp benchmarks show very similar trends. From the results, we see how, when the execution time achieved by regulating concurrency with SAC-STM is reduced, the corresponding values of WRMS looks very reduced. This tendency is noted independently of the amount of hidden nodes, as soon as at least a minimum amount of iterations of the learning algorithm are carried out. Also, the value of WRMS tends to decrease vs the number of iterations of the learning algorithm. The evident exception is noted for the case of 32 hidden-nodes for the intruder benchmark, where some spikes are observed for both the benchmark execution time and WRMS, in correspondence to some non-minimal values for the number of iterations. This may be attributed to over-fitting phenomena that may arise when the number of hidden nodes in NN is oversized. On the other hand, with undersized values for the number of hidden-nodes, such as with 4 hidden nodes, the value of WRMS tends to decrease slowly, which lead concurrency regulation to become less effective in reducing the actual execution time for the benchmark. However, the data show that for configurations with reasonable amounts of hidden nodes, the reliability of WRMS as the means for expressing the re-

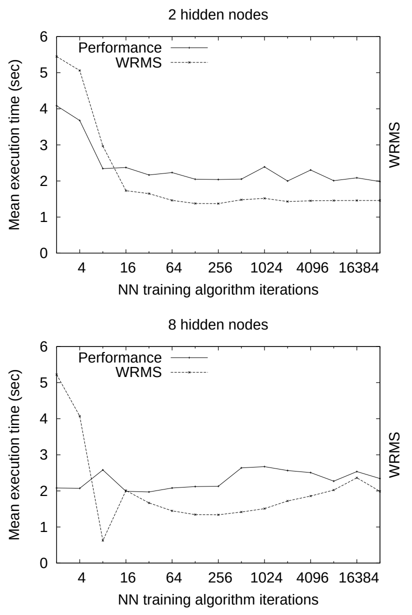
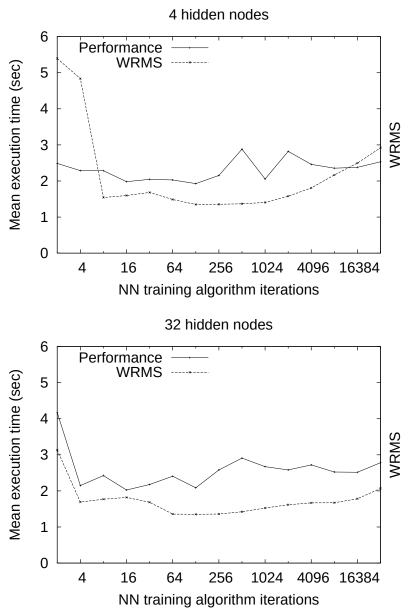


Figure 5.24: Relation between the final achieved performance and WRMS for vacation

lation between the quality of the waste of time prediction by NN and the final performance achieved while regulating concurrency on the basis of that prediction is actually assessed. This occurs even for reduced values of the number of iterations.

5.5.4 The Actual Dynamic Feature Selection Architecture

To support dynamic selection of relevant features to be sampled and exploited for concurrency regulation, we need to rely on a set of NN instances (not a unique instance as instead it occurs in SAC-STM), each one able to manage a different feature-set and properly trained on that set. These NN instances can be trained in parallel during the early phase of application processing. Then a so called Parameter-Scaling-Algorithm (PSA), implemented within an additional module

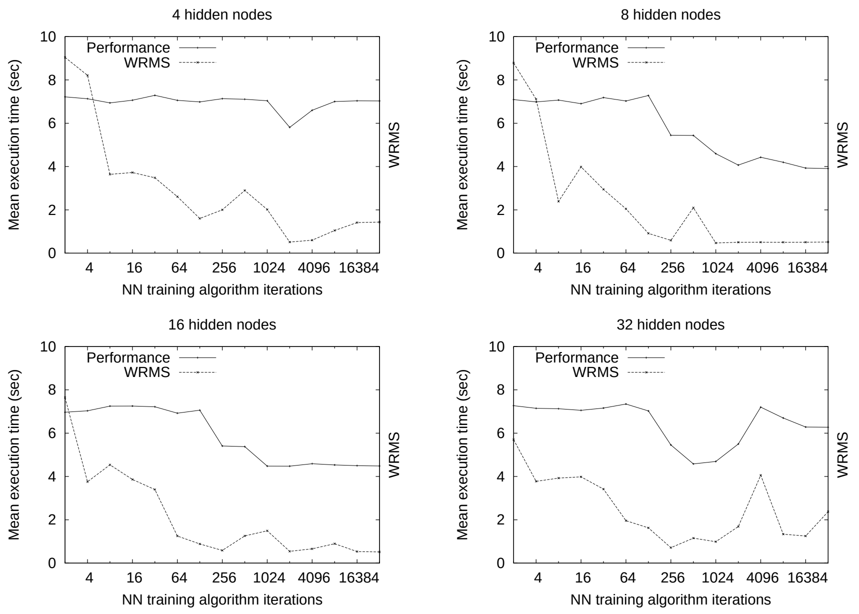


Figure 5.25: Relation between the final achieved performance and WRMS for intruder

integrated in SAC-STM, can be exploited for dynamically scaling-up/down the set of features (also referred to as parameters in the final architecture) to be taken into account for concurrency regulation along the sub-sequent execution window. Thus PSA is aimed at determining the NN instance to be used in relation to the selected parameters' set. The schematization of the architecture entailing dynamic feature selection capabilities is shown in Figure 5.26.

To select the best suited NN instance, and hence the sub-set of features that can be considered as reliable representative of the workload actual behavior, PSA performs the following tasks: (1) It periodically (e.g. at the end of the observation window) evaluates the quality of the prediction by the currently in use instance of NN (representative of the currently in use set of features) via estimation of WRMS; (2) It periodically analyzes the statistics related to the currently monitored features, to determine variance and correlation.

If the calculation of WRMS in point 1 gives rise to a value exceeding a specific threshold, then PSA enlarges the set of input features, to be exploited for concurrency regulation in the subsequent observation window, to the maximum set formed by the 6 features originally used in SAC-STM, namely $maxSet = \{rs_s, ws_s, rw_a, ww_a, t_t, ntc_t\}$. It then issues commands to SC, CA and NN in order to trigger their internal reconfiguration, leading to work with $maxSet$. This means that any query from CA while performing concurrency regulation during the subsequent observation window needs to be answered by relying on the NN instance trained over $maxSet$.

On the other hand, in case the WRMS value computed in point 1 does not exceed the threshold value, the analysis in point 2 is exploited to determine whether the currently in use set of features can be shrunk (and hence to determine whether a scale-down of the set of sampled parameters can be actuated). Particularly, if the variance observed for a given feature is lower than a given threshold, the feature is discarded from the relevant feature-set to be exploited in the next observation window. Then for each couple of not yet discarded features, PSA calculates their correlation and, if another threshold is exceeded, one of them is discarded too. The non-discarded features form the optimized (shrunk) set of parameters to be exploited for concurrency regulation in the subsequent observation window, which we refer to as $minSet$. Then, similarly to what done before, PSA issues configuration commands to SC, CA and NN in order to trigger them for operating with $minSet$.

The values that have been used for configuring threshold parameters and the length of the observation window for the actual experimentation of the final architecture, whose outcomes are reported in the next section, have been

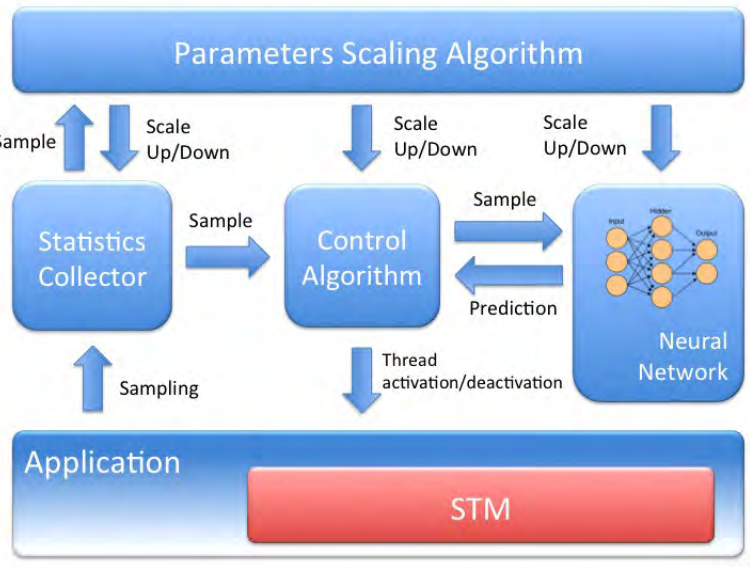


Figure 5.26: Extended SAC-STM architecture

selected on the basis of empirical observations. Procedures for automatizing the configuration are planned as future work along this same research path. Finally, in the new architecture in Figure 5.26, all the tasks associated with concurrency regulation, which are performed by the modified versions of SC, CA and NN, and by the added PSA module, are carried out off the application critical path. Specifically, they are executed along different, low priority threads, which spend most of their time in the waiting state. Hence, intrusiveness of these threads is extremely limited, as we will also show in Section 5.6 via experimental data.

5.6 Experimental Evaluation

In this section we present the results of an experimental study aimed at evaluating the effectiveness of our proposal. The provided data are related to experiments carried out by still running applications from the STAMP benchmark

variance thresholds				
	ssca2	**intruder**	**vacation**	**mod. vacation**
rs_s	4	50	600	340
ws_s	2	3	16	3
rw_a	$5 \cdot 10^{-5}$	0,018	$6 \cdot 10^{-6}$	10^{-4}
ww_a	$5 \cdot 10^{-5}$	$2,5 \cdot 10^{-5}$	$1,2 \cdot 10^{-4}$	10^{-4}
correlation thresholds				
	ssca2	**intruder**	**vacation**	**mod. vacation**
all param.	0,85	0,85	0,85	0,85
variance and correlation analysis window				
	ssca2	**intruder**	**vacation**	**mod. vacation**
#transactions	$4 \cdot 10^5$	$4 \cdot 10^5$	$2 \cdot 10^5$	$4 \cdot 10^5$
concurrency regulation interval				
	ssca2	**intruder**	**vacation**	**mod. vacation**
#transactions	4000	4000	4000	4000

Table 5.3: Parameters configuration for the performance tests

suite on top of the 16-cores HP ProLiant machine that has been exploited for the previous reported experiments. As for STAMP, we selected three applications, namely intruder, ssca2 and vacation, since they exhibit quite different execution profiles. Further, we provide experimental data in relation to a modified version of vacation, properly configured to stress (and thus further evaluate) the innovative capabilities by the architecture deriving from our proposal. Note that now each adaptive STM is configured so that each thread collects statistics used to build the samples that characterize the transactional workload. In Table 5.3 we list the statically configured values for the platform parameters, which have been used for the experiments.

In Figure 5.27 we show the results achieved with the intruder benchmark. In particular, we report the benchmark execution time while varying the number of CPU-cores allowed to be used for application execution. This is reflected into a maximum value for the number of threads running the application. In fact, in our study we adhere to the common practice of avoiding the usage

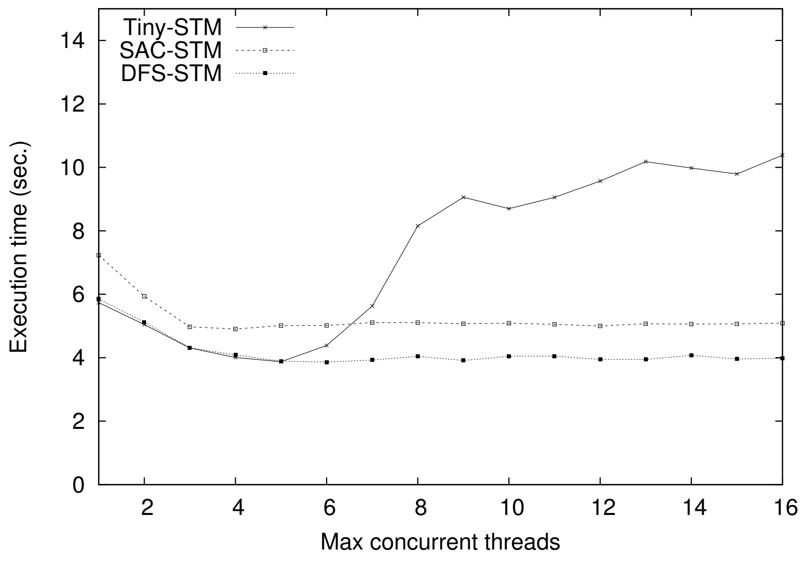


Figure 5.27: Results for intruder

of more application threads than the available CPU-cores, which is done in order to avoid suboptimal execution scenarios for STM systems characterized by excessive context-switch overhead [66]. We note that the original version of TinySTM always exploits all the allowed to be used CPU-cores, since it does not entail any concurrency regulation scheme. On the other hand, both SAC-STM (which is taken as a reference together with TinySTM) and the new architecture we have provided, which we refer to as Dynamic-Feature-Selection STM (DFS-STM) in the rest of this study, perform concurrency regulation. Hence, they both lead the application to use a variable amount of threads over time, which ranges from 1 to the maximum value admitted for the specific experimentation point. By the data, the TinySTM curve shows that the benchmark reaches its minimum execution time with static concurrency level set to 5. Beyond this value, data contention brings the application to pay large penalties caused by excessive transaction rollback. Thus, the performance delivered by TinySTM

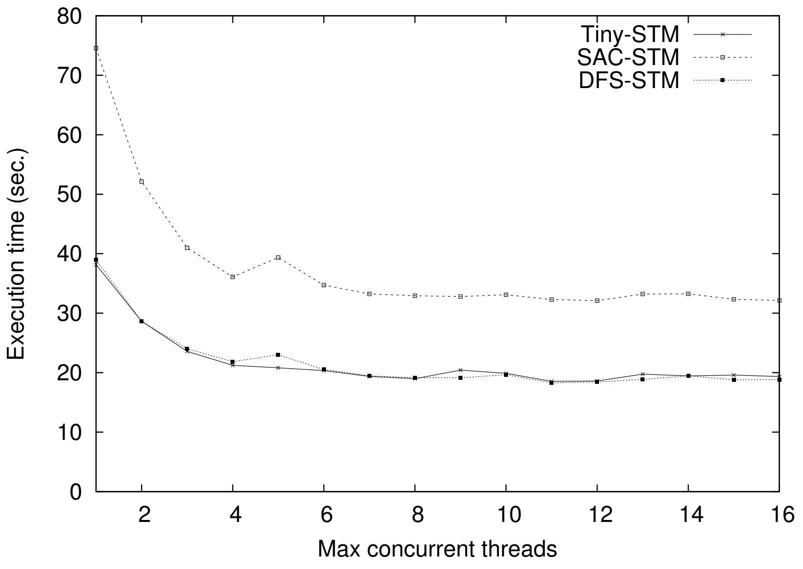


Figure 5.28: Results for ssca2

rapidly decreases when running the application using more than 5 CPU-cores. Conversely, SAC-STM and DFS-STM provide the same performance achievable with the optimal degree of concurrency for any value of the maximum number of threads in the interval [5-16]. Hence, they correctly regulate concurrency to the optimal level, even when more CPU-cores are available. However, by dynamically shrinking the set of input features to be sampled, DFS-STM allows up to 30% reduction of the benchmark execution time. Hence, it reveals effective in significantly reducing the overhead associated with the static feature-selection approach used by SAC-STM.

The performance data for ssca2, reported in Figure 5.28, look somehow different. Particularly, the TinySTM curve reveals that no thrashing occurs, even when running the application by relying on all the 16 available CPU-cores. This means that, for this benchmark, concurrency regulation cannot be expected to improve performance significantly. However, the results show SAC-STM pays a

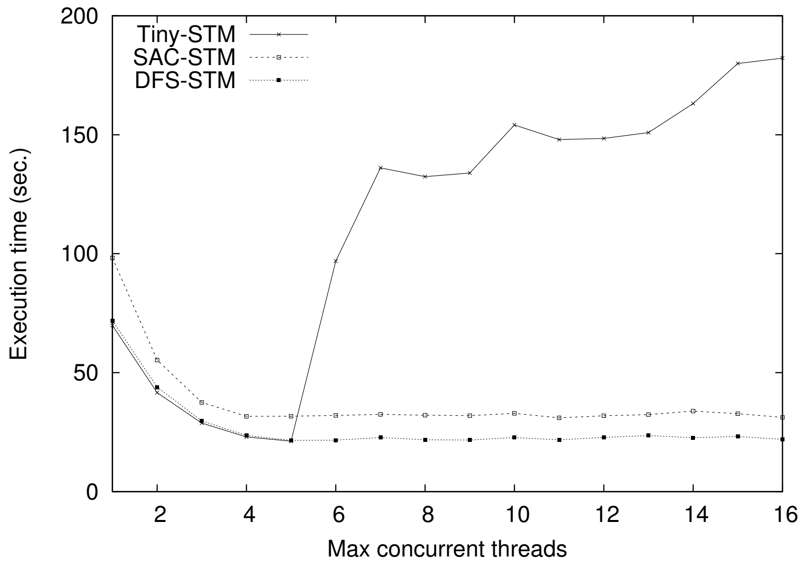


Figure 5.29: Results for vacation

relevant sampling cost (with no particular revenue from the concurrency regulation process, as hinted above), which leads it to deliver performance from 87% to 60% worse than the one delivered by TinySTM, depending on the maximum allowed number of concurrent threads. Such an overhead is fully removed by DFS-STM, which allows delivering the same identical performance as TinySTM (still with no advantage from concurrency regulation). We note that the overhead reduction, beyond indicating the effectiveness of dynamically shrinking the set of features to be sampled, also indicates null intrusiveness of the additional tasks performed by DFS-STM, such as the execution of PSA.

Figure 5.29 shows the results for the standard version of vacation. Also in this case we reach the optimum performance with 5 threads. Beyond this value, TinySTM exhibits rapidly decreasing performance, just for the reasons explained above. Again, SAC-STM and DFS-STM allow regulating the concurrency level to the best suited value. However, SAC-STM shows significant overhead. Hence, DFS-STM reduces the execution time by about 30%.

All the benchmark configurations that have been considered so far are characterized by phase-based execution profiles, with very few changes along the execution. In order to evaluate DFS-STM with highly dynamic workloads, the modified version of vacation has been exploited. Essentially, vacation emulates a travel reservation system, where customers can reserve flights, rooms and cars. The fraction of transactions accessing each one of the three types of items is fixed over time. This is representative of scenarios where the popularity of the different types of items does not change over time. We modified this feature in order to emulate scenarios where the item popularity can show significant changes according to a periodic basis. We note that this kind of scenarios are prone to take place in relation to real-life events (e.g. associated with relevant promotional sales or new product launches). In the modified version of vacation, the fractions of transactions accessing the three types of items periodically changes. Specifically, the fraction of transactions accessing car-items changes over time according to the curve depicted in Figure 5.30. The remaining fraction is equally split into transactions accessing flight and room items.

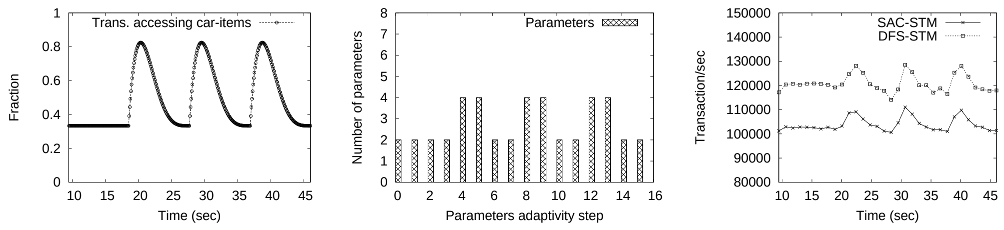


Figure 5.30: Parameters and throughput variation over time for the modified vacation benchmark

For this workload, we show in Figure 5.30 how the number of input features, selected as relevant by DFS-STM, changes over time. These results refer to an

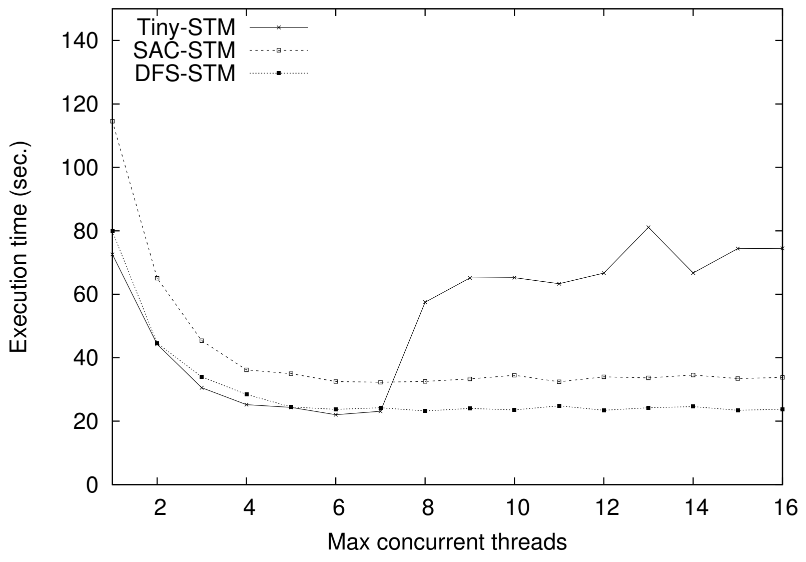


Figure 5.31: Results for modified vacation

execution where we allow a maximum number of concurrent threads equal to 8. We note that, whenever the mix of transactions remains quite constant over time (e.g. up to 17 seconds of the execution, or in the interval between 22 and 27 seconds of the execution), only two parameters (specifically t_t and ntc_t) are selected. Conversely, whenever the mix of transactions rapidly changes (e.g. in the interval between 17 and 22 seconds of the execution, or between 27 and 32 seconds), which leads to increase the variance and/or un-correlation of some workload features, the number of parameters grows to 4 (specifically including t_t, ntc_t, ws_s, rs_s).

The throughput achieved with both SAC-STM and DFS-STM is shown on the right of Figure 5.30. Also in this case DFS-STM achieves a remarkable performance improvement with respect to SAC-STM. For completeness, the execution time while varying the maximum number of concurrent threads for the case of the modified vacation benchmark is depicted in Figure 5.31, which

	ssca2	**intruder**	**genome**	**kmeans**	**yada**	**vacation**	**labyrinth**	**bayes**
DFS-STM	25%	18%	54%	23%	23%	40%	32%	51%
TinySTM	25%	09%	32%	18%	19%	7%	23%	33%

Table 5.4: Minimum achieved percentage of ideal speedup

again shows the relevant gain that can be achieved by DFS-STM over SCA-STM thanks to the reduction of the overhead for supporting concurrency regulation. Finally, in Table 5.4 we report the minimum percentage of the ideal speedup (over serial execution of the same application on a single CPU-core) which is achieved by DFS-STM and by TinySTM when considering variations of the number of CPU-cores between 1 and 8. For all the applications of the STAMP suite DFS-STM generally guarantees much higher percentage values of the ideal speedup, which again indicates its ability to efficiently control the parallelism degree by both avoiding thrashing phenomena and inducing very reduced feature sampling overhead.

Chapter 6

The Analytical Model Based Approach

In this chapter we tackle the issue of regulating the concurrency level in STM via a model-based approach, which differentiates from classical ones in that it avoids the need for the STM system to meet specific assumptions (e.g. in terms of data access pattern). Our proposal relies on a parametric analytical expression capturing the expected trend in the transaction abort probability (versus the degree of concurrency) as a function of a set of features associated with the actual workload profile. The parameters appearing within the model exactly aim at capturing execution dynamics and effects that are hard to be expressed through classical (non-parametric) analytical modelling approaches. We derived the parametric expression of the transaction abort probability via combined exploitation of literature results in the field of analytical modelling and a simulation-based analysis. Further, the parametric model is thought to be easily customizable for a specific STM system by calculating the values to be assigned to the parameters (hence by instantiating the parameters) via regression analysis. The latter can be performed by exploiting a set of sampling data gathered through run-time observations of the STM application. However, differently from what happens for the training process in some machine learning approaches, the actual sampling phase (needed to provide the knowledge base for regression in our approach) is very light. Specifically, a very limited number of profiling samples, related to a few different concurrency levels for the STM system, likely suffices for successful instantiation of the model parameters via regression. Finally, our approach inherits the extrapolation capabilities proper of pure analytical models (although it does not require their typical stringent assumptions to be met, as already pointed out), hence allowing reliable performance forecast even for concurrency levels standing distant from the ones for which sampling was actuated. A bunch of experimental results achieved by running the STAMP benchmark suite on top of the TinySTM open source framework are reported for validating the proposed modelling approach. Further, we present the implementation of a concurrency self-regulating STM, exploiting the proposed performance model, still relying on TinySTM as the core STM layer, and we report experimental data for an assessment of this architecture.

6.1 The Parametric Performance Model

As already hinted, we decided to exploit a model relying on a parametric analytical expression which captures the expected trend of the transaction abort probability as a function of (1) a set of features characterizing the current workload profile, and (2) the number of concurrent threads sustaining the STM application. The parameters in the analytical expression aim at capturing effects that are hard to express through a classical (non-parametric) analytical modelling approach. Further, they are exploited to customize the model for a specific STM application through regression analysis, which is done by exploiting a set of sampling data gathered through run-time observations of the application. In the remainder of this section we provide the basic assumptions on the behaviour of the STM application, which are exploited while building the parametric analytical model. Then the actual construction of the model is presented, together with a model validation study.

6.1.1 Basic Assumptions

The STM application is assumed to be run with a number k of concurrent threads. The execution flow of each thread is characterized by the interleaving of transactions and non-transactional code (*ntc*) blocks. This is the typical structure for common STM applications, which also reflects the one of widely diffused STM benchmarks (see, e.g., [20]). The transaction read-set (write-set) is the set of shared data-objects that are read (written) by the thread while running a transaction. If a conflict between two concurrent transactions occurs, then one of the conflicting transactions is aborted and re-started (which leads to a new transaction run). After the thread commits a transaction, it executes a *ntc* block, which ends before the execution of the begin operation of the subsequent transaction along the same thread.

6.1.2 Model Construction

The set P of features exploited for the construction of the parametric analytical model, which are used to capture the workload profile, are the ones already described in section 4.3.

Our parametric analytical model expresses the transaction abort probability p_a as a function of the features belonging to the set P, and the number k of concurrent treads supposed to run the STM application. Specifically, it instantiates (in a parametric manner) the function

$$p_a = f(rs_s, ws_s, rw_a, ww_a, t_t, ntc_t, k) \tag{6.1}$$

Leveraging literature models proposing approximated performance analysis for transaction processing systems (see [73, 74]), we express the transaction abort probability p_a through the function

$$p_a = 1 - e^{-\alpha} \tag{6.2}$$

However, while in literature the parameter α is expressed as the multiplication of parameters directly representing, e.g., the data access pattern and the workload intensity (such as the transaction arrival rate λ for the case of open systems), in our approach we express α as the multiplication of different functions that depend on the set of features appearing in equation (6.1). Overall, our expression for p_a is structured as follows

$$p_a = 1 - e^{-\rho \cdot \omega \cdot \phi} \tag{6.3}$$

where the function ρ is assumed to depend on the input parameters rs_s, ws_s, rw_a and ww_a, the function ω is assumed to depend on the parameter k, and the function ϕ is assumed to depend on the parameters t_t and ntc_t.

We note that equation (6.2) has been derived in literature while modeling the abort probability for the case optimistic concurrency control schemes, where transactions are aborted (and restarted) right upon conflict detection. Consequently, this expression for p_a and the variation we propose in equation (6.3) are expected to match the STM context, where pessimistic concurrency control schemes (where transactions can experience lengthy lock-wait phases upon conflicting) are not used since they would limit the exploitation of parallelism in the underlying architecture. More specifically, in typical STM implementations (see, e.g., [19]), transactions are immediately aborted right upon executing an invalid read operation. Further, they are aborted on write-lock conflicts either immediately or after a very short wait-time.

The model we propose in equation (6.3) is parametric thanks to expressing α as the multiplication of parametric functions that depend on a simple and concise representation of the workload profile (via the features in the set P) and on the level of parallelism. This provides it with the ability to capture variations of the abort probability (e.g. vs the degree of parallelism) for differentiated application profiles. Particularly, different applications may exhibit similar values for the featuring parameters in the set P, but may anyhow exhibit different dynamics, leading to a different curve for p_a while varying the degree of parallelism. This is catchable by our model via application-specific instantiation of the parameters characterizing the functions ρ, ω and ϕ, which can be done through regression analysis. In the next section we discuss how we have derived the actual ρ, ω and ϕ functions, hence the actual function expressing α.

6.1.3 Instantiating ρ, ω and ϕ

The shape of the functions ρ, ω and ϕ determining α is derived in our approach by exploiting the results of a simulation study. We decided to rely on simulation, rather than using measurements from real systems, since our model is aimed at capturing the effects associated with data contention on the abort probability, while it is not targeted at capturing the effects of thread-contention on hardware resources. Consequently, the instantiation of the functions appearing within the model has been based on an “ideal hardware” simulation model showing no contention effects. Anyway, when exploiting our data contention model for concurrency regulation in a real system, a hardware scalability model (e.g. a queuing network-based model) can be used to estimate variations of the processing time, due to contention effects on shared hardware resources, as a function of the number of the concurrent threads. In the final part of this chapter, we provide some results that have been achieved by exactly using our data contention model and a hardware scalability model in a joint fashion. The simulation framework we have exploited in this study is the same used in [14] for validating an analytical performance model for STM. It relies on the discrete-event paradigm, and the implemented model simulates a closed system with k concurrent threads, each one alternating the execution of transactions and *ntc* blocks. The simulated concurrency control algorithm is the default algorithm of TinySTM (encounter time locking for write operations and timestamp-based read validation). A transaction starts with a *begin* operation, then it interleaves the execution of read/write operations (accessing a set of shared data objects) and local computation phases, and, finally, executes a *commit* operation. The durations of *ntc* blocks, transactional operations and local computation phases are exponentially distributed.

In the simulation runs we performed to derive and validate the expression of α, we varied rs_s and ws_s between 0 and 200, rw_a and ww_a between $25 \cdot 10^{-6}$ and 0.01, t_t between 10 and 150 μsec, and ntc_t between 0 and $15 \cdot 10^4$ μsec. These intervals include values that are typical for the execution of STM benchmarks such as [20], hence being representative of workload features that can be expected in real execution contexts. Further, we varied k between 2 and 64 in the simulations. We omit to explicitly show all the achieved simulation results, but the showed ones are a significative, although concise, representation of the whole set of achieved results.

The construction of the analytical expressions for ρ, ω and ϕ has been based on an incremental approach. Particulary, we first derive the expression of ρ analyzing simulation results while varying workload configuration parameters affecting it, i.e. rs_s, ws_s, rw_a, ww_a, and keeping fixed other parameters. After, we calculate the values of ρ from the ones achieved for p_a via simulation, which is done by using the inverse function $\rho = f^{-1}(p_a)$, once set $\omega = 1$ and $\phi = 1$. After having identified a parametric fitting function for ρ, we derive the expression of ω via the analysis of the simulation results achieved while also varying k. Hence, we calculate $\omega = f^{-1}(p_a)$, where we use for ρ the previously identified expression, and where we set $\phi = 1$. Therefore, we select a parametric fitting function for ω. Finally, we use the same approach to derive the expression of ϕ, which is done by exploiting the simulation results achieved while varying all the workload profile parameters and the level of concurrency k, thus calculating $\phi = f^{-1}(p_a)$, where we use for ρ and ω the previously chosen expressions.

In order to derive the expression of ρ, we initially analyzed via simulation the relation between the values of p_a and the values of the parameters ws_s and ww_a. In Figure 6.1 we provide some results showing the values of ρ as calculated through the $f^{-1}(p_a)$ inverse function (like depicted above) by relying on simulation data as the input. The data refer to variations of ww_a and to 3 different values of ws_s, while all the other parameters have been kept fixed. We note that ρ appears to have a logarithmic shape. Additionally, in order to chose a parametric function fitting the calculated values of ρ, we need to consider that if $ww_a = 0$ then $p_a = 0$. In fact, no data contention ever arises in case of no write operations within the transactional profile (which implies $\rho = 0$). Thus,

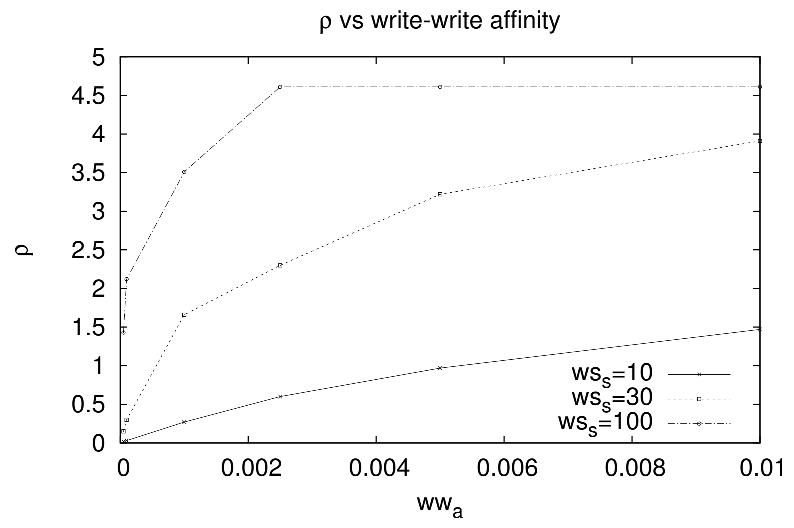


Figure 6.1: Variation of ρ with respect to the write-write affinity

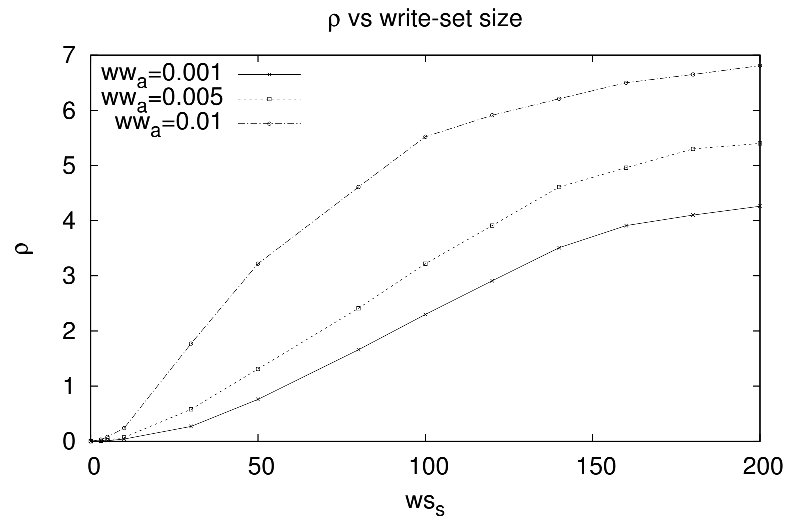


Figure 6.2: Variation of ρ with respect to the write-set size

we approximated the dependency of ρ on ww_a through the following parametric logarithmic function

$$c \cdot ln(a \cdot ww_a + 1) \tag{6.4}$$

where a and c are the fitting parameters. The presence of the $+1$ term in expression (6.4) is due to the above-mentioned constraint according to which $ww_a = 0$ implies $\rho = 0$.

After, we also considered the effects of the parameter ws_s on ρ. To this aim, in Figure 6.2 we report the values of ρ, derived from the simulation results, while varying ws_s and for 3 different values of ww_a. We remark the presence of a flex point. Therefore, in this case, we approximated the dependency of ρ on ws_s by using the function

$$e \cdot (ln(b \cdot ws_s + 1))^d \tag{6.5}$$

where b, d and e are fitting parameters, d being the one capturing the flex. Assuming that the effects on the transaction abort probability are multiplicative with respect to ww_a and ws_s (which is aligned to what literature models state in term of the proportionality of the abort probability wrt the multiplication of the conflict probability and the number of operations, see, e.g., [73]), we achieved the following parametric expression of ρ (vs ww_a and ws_s), where d has been used as the exponent also for expression (6.4) in order to capture the effects of shifts of the flex point caused by variations of ww_a (as shown by the plots in Figure 6.2 relying on simulation)

$$[c \cdot (ln(b \cdot ws_s + 1)) \cdot ln(a \cdot ww_a + 1)]^d \tag{6.6}$$

where we collapsed the original parameters c and e within one single parame-

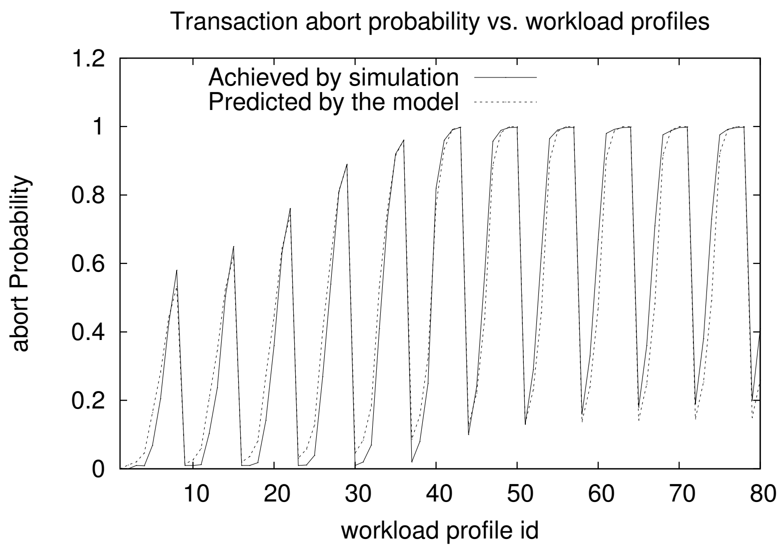


Figure 6.3: Simulated vs predicted abort probability while varying ww_a and ws_s

ters c. We validated the accuracy of the expression (6.6) via comparison with values achieved through a set of simulations, where we used different workload profiles. The parameters appearing in expression (6.6) have been calculated through regression analysis. Specifically, for each test, we based the regression analysis on 40 randomly selected workload profiles achieved while varying ww_a and ws_s. Then, we measured the average error between the transaction abort probability evaluated via simulation and the one predicted using for ρ the function in expression (6.6) for a set of 80 randomly selected workload profiles. As an example, in Figure 6.3, we depict results for the case with $k = 8$. Along the x-axis, workload profiles are identified by integer numbers and are ordered by the values of ws_s and ww_a. The measured average error in all the tests was 5.3%.

Successively, we considered the effects on the transaction abort probability caused by read operations. Thus, we analyzed the relation between p_a and the

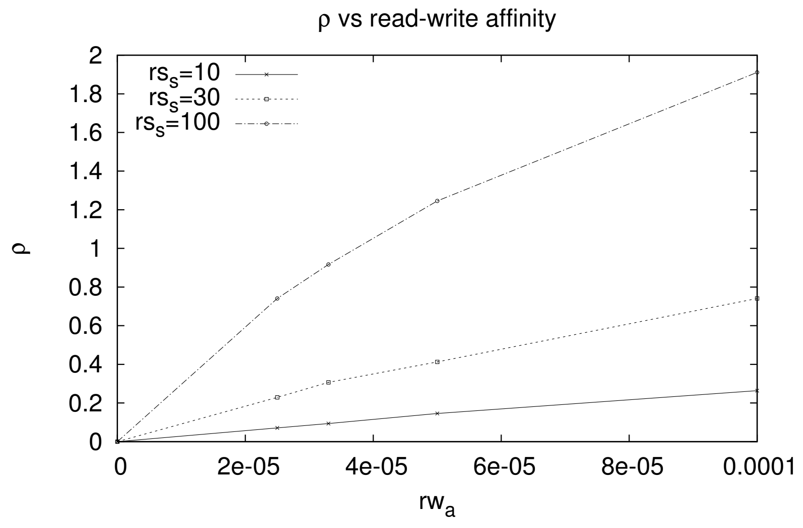


Figure 6.4: Variation of ρ with respect to the read-write affinity

parameters rs_s, rw_a and ws_s. The parameter ws_s is included since contention on transactional read operations is affected by the amount of write operations by concurrent transactions. In Figure 6.4 we report simulation results showing the values of ρ while varying rw_a and for 3 different values of rs_s. In Figure 6.5, we report values of ρ achieved while varying rs_s and for 3 different values of rw_a. We note that the shape of the curves are similar to the above cases, where we analyzed the relation between p_a and the parameters ww_a and ws_s. Thus, using a similar approach, and considering that p_a is also proportional to ws_s, we approximate the dependency of ρ on rw_a, ws_s and ww_a using the following function

$$[e \cdot (ln(f \cdot rw_a + 1)) \cdot ln(g \cdot rs_s + 1) \cdot ws_s]^z \tag{6.7}$$

where e, f, g and z are the fitting parameters. The final expression for ρ is then derived summing expressions (6.6) and (6.7). The intuitive motivation is

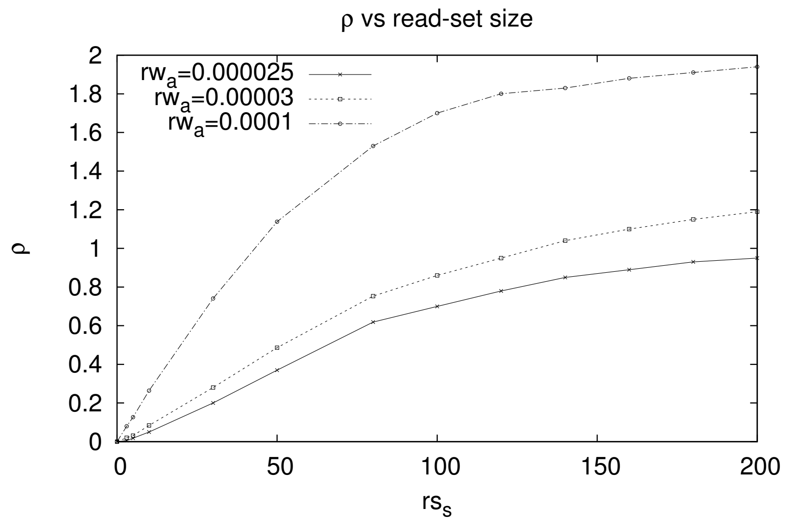


Figure 6.5: Variation of ρ with respect to the read-set size

that adding read operations within a transaction, the likelihood of abort due to conflicts on original write operations does not change. However, the added operations lead to an increase of the overall abort probability, which we capture summing the two expressions. Also in this case, we validated the final expression for ρ via comparison with the values achieved through a set of simulations, where we varied the workload profile. Similarly to what was done before, the regression analysis has been based on 40 workload profiles, while the comparison has been based on 80 workload profiles, all selected by randomly varying ww_a, ws_s, rw_a, rs_s. The results for $k = 8$ are reported in Figure 6.6. Along the x-axis, workload profiles are ordered by values of rs_s, rw_a, ws_s and ww_a. The average error we measured in all the tests was 2.7%.

Successively, in order to build the expression for ω, we considered the effects of the number of concurrent threads, namely the parameter k, on the abort probability. On the basis of simulation results, some of which are reported in

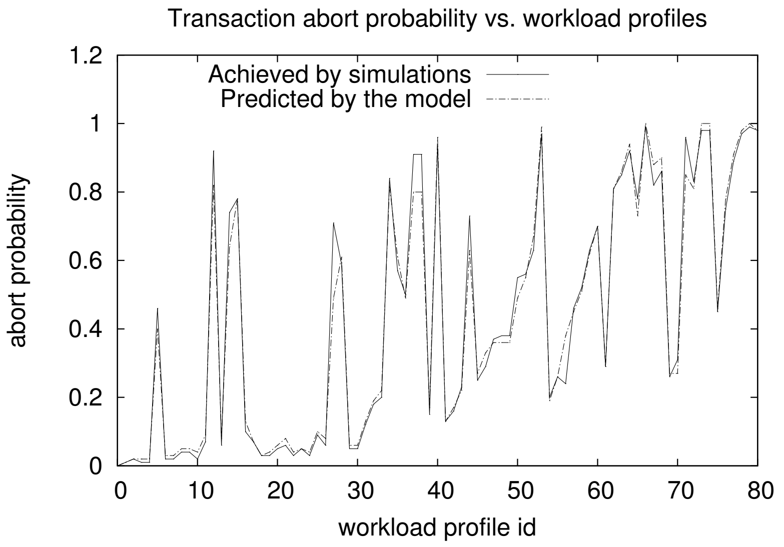


Figure 6.6: Simulated vs predicted abort probability vs rw_a, rs_s, ww_a, ws_s

Figure 6.7, we decided also in this case to use a parametric logarithmic function as the approximation curve of ω vs k. Clearly, the constraint needs to be accounted for that if $k = 1$ then $\omega = 0$ (since the absence of concurrency cannot give rise to transaction aborts). Thus, we approximate ω as

$$h \cdot (ln(l \cdot (k-1) + 1), \tag{6.8}$$

where h and l are the fitting parameters. Again, we validated the out-coming function for p_a, depending on ω (and hence depending on modeled effects of the variation of k), using the same amount of workload profiles as in the previous studies, still selected by randomly varying ww_a, ws_s, rw_a, rs_s and k. Some results are depicted in Figure 6.8 for variations of k between 1 and 64. The average error we measured in all the tests was 2.1%.

Finally, we built the expression of ϕ, which depends on t_t and t_{ntc}. To

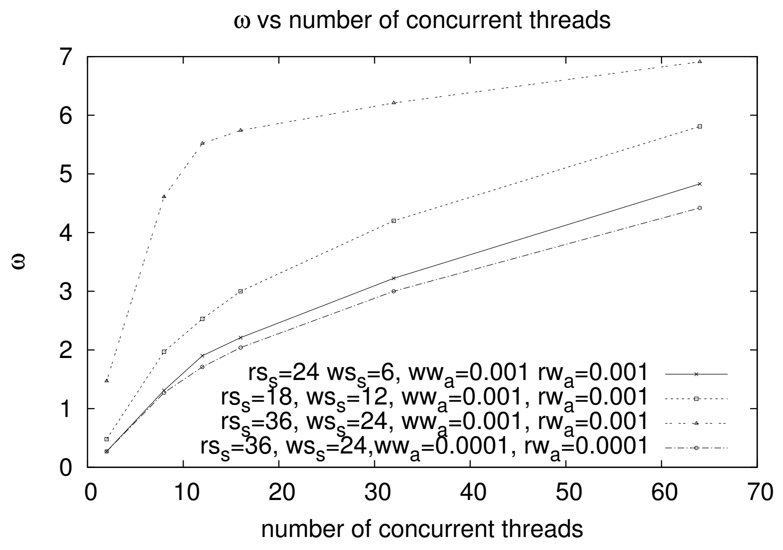


Figure 6.7: Variation of ω vs the number of concurrent threads

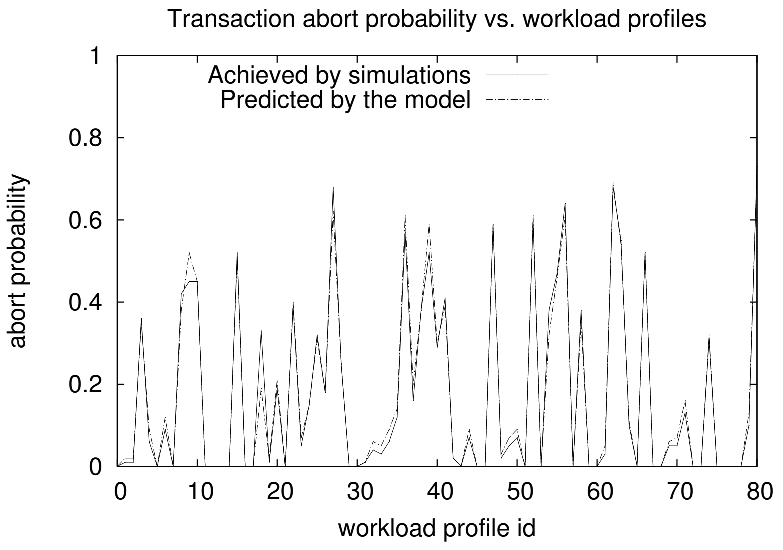


Figure 6.8: Simulated vs predicted abort probability vs k, rw_a, rs_s, ww_a, ws_s

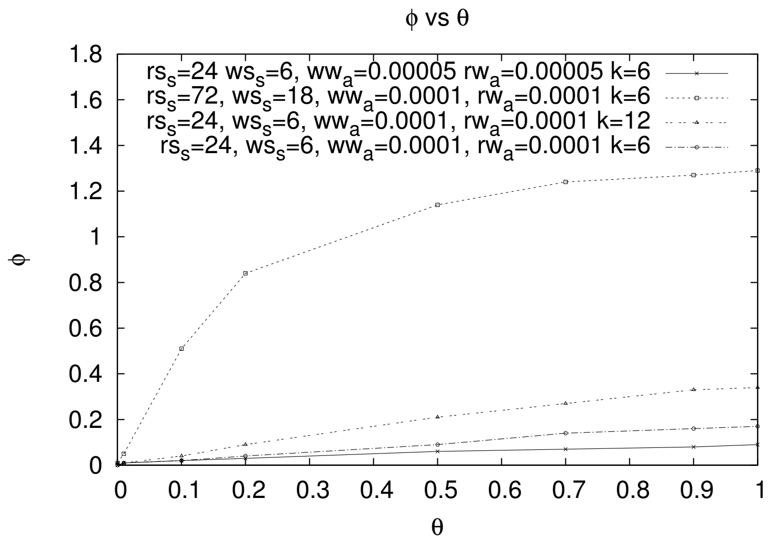


Figure 6.9: Variation of ϕ with respect to θ

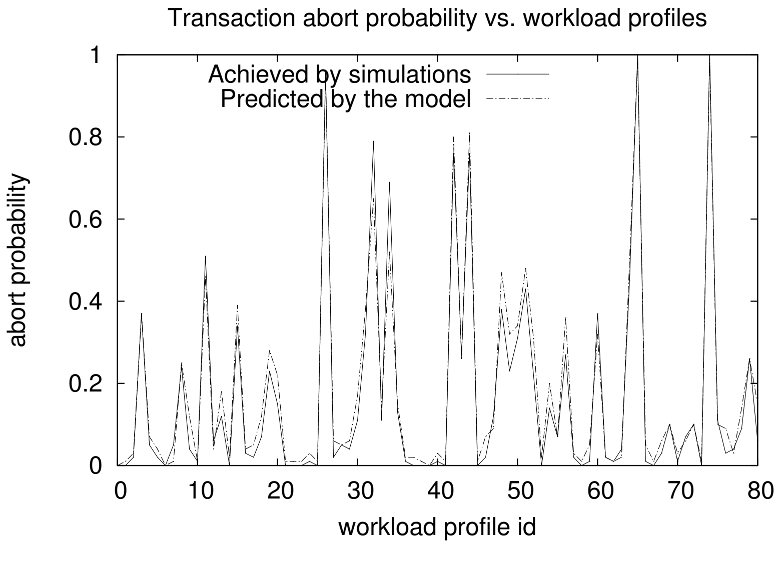


Figure 6.10: Simulated vs predicted abort probability while varying all the workload profile parameters

this aim, we note that if $t_t = 0$ (which represent the unrealistic case where transactions are executed instantaneously) then ϕ must be equal to 0 (given that the likelihood of concurrent transactions is zero). Additionally, we note that t_t can be seen as the duration of a *vulnerability window* during which the transaction is subject to be aborted. For longer fractions of time during which transactions are vulnerable, higher probability of actual transaction aborts can be expected. Thus we assume ϕ to be proportional to

$$\theta = \frac{t_t}{t_t + ntc_t} \tag{6.9}$$

We analyzed through simulation the relation between ϕ and θ. Some results are shown in Figure 6.9, on the basis of which we decided to approximate ϕ using the function:

$$m \cdot ln(n \cdot \theta + 1) \tag{6.10}$$

where m and n are the fitting parameters.

The expression of p_a in equation (6.3) is now fully defined. To validate it, we used the same approach that has been adopted for the validation of each of the aforementioned incremental steps. Some results, where we randomly selected workload profiles, are shown in Figure 6.10. In all our tests, we measured an average relative error of 4.8%.

6.1.4 Model Validation with Respect to a Real System

As a further validation step we compared the output by the proposed model with real measurements taken by running applications belonging to the STAMP benchmark suite on top of the open source TinySTM framework. Additionally, we evaluated the model ability to provide accurate predictions while varying the amount of samples used to perform the regression analysis, gathered through observations of the behavior of the real system. Particularly, we evaluated the extrapolation capability of the model, namely its ability to forecast the transaction abort probability that would be achieved when running the STM application with concurrency levels (number of threads) not included in the observed domain where regression samples were taken.

The presented results refer to three different benchmark applications of the STAMP suite, namely kmeans, yada and vacation. As shown in paragraph B and with more detail in [20], these applications are characterized by quite different workload profiles. This allowed us to evaluate the model accuracy with respect to a relatively wide workload configuration domain. All the tests have been performed on top of the platform described in section 2.2.

For each application, we performed regression analysis to calculate three different sets of values for the model parameters, hence instantiating three models relying on the proposed parametric analysis. Any regression has been performed using one of three different sets of measurements, each set including 80 samples. The first set included samples gathered observing the application running with 2 and 4 concurrent threads. The second one included samples gathered observing the application running with 2, 4 and 8 concurrent threads. Finally, the third one included samples gathered observing the application running with 2, 4, 8 and 16 concurrent threads. This allowed us to evaluate the extrapolation ability of the model, with respect to the number of concurrent threads, while observing the application for limited amounts of concurrency levels (say for 2, 3 or 4 different levels of concurrency). We performed, for each application, the following tests. After setting up the model instances, we executed a set of runs of the application using different values for the application input parameters (leading the same application to run with somehow different workload profiles) and with a number of concurrent threads spanning from 2 to 16. During each run, we measured the average values of the workload profile features included in the set P along different observation intervals having a pre-established length, and we used them as the input to the three instantiated models in order to compute the expected abort probability for each observation interval. After, for each instantiated model, we compared the predicted value with the real one observed during the runs.

In Table 6.1, we report the average value of the prediction error (and its variance) for all the target benchmark applications, and for the three model instances, while considering variations of the actual level of concurrency between 2 and 16. By the results, we note that, for the cases of yada and vacation, it has been sufficient to execute regression analysis with samples gathered observing the application running with only 2 and 4 threads in order to achieve an average prediction error bounded by 2.4% for any level of concurrency between 2 and 16. When enlarging the observation domain for the gathering of samples to be used by regression, i.e. when observing the application running also with 8 concurrent threads, we achieved for yada a slight error reduction. With vacation, the reduction is more accentuated. On the other hand, the prediction error achieved for kmeans with observations of the application running with 2 and 4 concurrent threads was greater. However, such an error drastically drops down when including samples gathered with 8 concurrent threads in the data set for regression. As for regression based on samples gathered with 2, 4, 8 and 16 threads, we note that the error marginally increases in all the cases. We believe that this is due to the high variance of the values of the transaction

	Observed concurrency levels for the regression analysis		
application	**2/4 threads**	**2/4/8 threads**	**2/4/8/16 threads**
vacation	2.166% (0,00089)	1.323% (0,00028)	1.505% (0,00032)
kmeans	18.938% (0,09961)	2.086% (0,00100)	2.591% (0,00109)
yada	2.385% (0,00029)	2.086% (0,00016)	2.083% (0,00022)

Table 6.1: Abort probability prediction error (and its variance)

abort probability we measured for executions with 16 concurrent threads, which gives rise to variability of the results of the regression analysis depending on the set of used observations. Overall, by the results showed in table 6.1, we achieved good accuracy and effectiveness by the model since it can provide low prediction error, for a relatively wide range of hypothesized thread concurrency levels (namely between 2 and 16) by just relying on observing the application running with 2, 4 and (at worst also) 8 concurrent threads.

We conclude this paragraph comparing the extrapolation ability of our model with respect to the neural network-based model proposed in chapter 5. To perform fair comparison, a same set of observations has been provided in input to both the models. Particularly, the reported results refer to the yada benchmark application, for which we provided a set of 80 observations (the same used for validating the model, as shown above), related to executions with 2 and 4 concurrent threads, in input to both our parametric model and the neural network based model presented in Chapter 5. As for the neural network approach, we used a back-propagation algorithm [68], and we selected the best trained network, in terms of prediction accuracy, among a set of networks having a number of hidden nodes spanning from 2 to 16, using a number of algorithm iterations spanning from 50 to 1600. In Figure 6.11, we show two dispersion charts, each one representing the correlation between the measured values of the transactional wasted time and the ones predicted using the model (left chart) and the

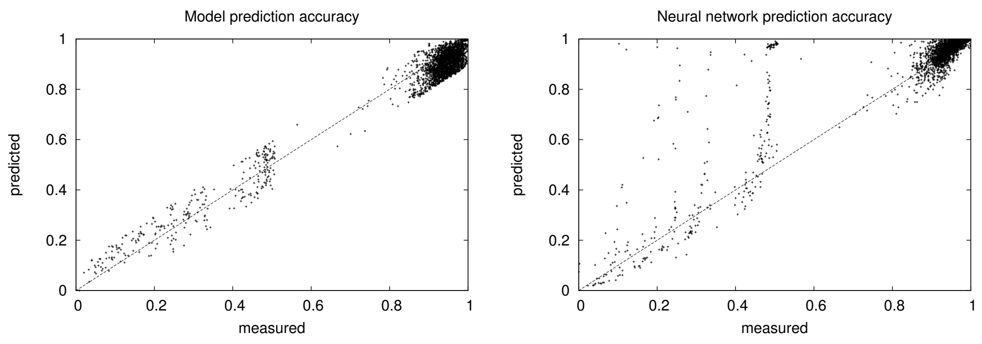


Figure 6.11: Model and neural-network prediction accuracy

neural network (right chart) (the transactional wasted time prediction process based on the model will be presented and discussed in Section 6.2). These refer to concurrency levels spanning in the whole interval 2-16. We remark that a lower prediction error corresponds to a higher concentration of points along the diagonal straight line evidenced in the graphs. We can see that, in the case of the neural network, there is a significantly wider dispersion of points compared to the model we are proposing. In fact, the average prediction error for the neural network is equal to 17.3%, while for the model it is equal to 2.385%. This is a clear indication of higher ability to extrapolate the abort probability by the model when targeting concurrency levels for which no real execution sample is available (and/or that are far from the concurrency levels for which sampling has been actuated). As a reflection, the parametric model we present provides highly reliable estimations, even with a few profiling data available for the instantiation of its parameters. Hence it is suited for the construction of concurrency regulation systems inducing low overhead and providing timely selection of the best suited parallelism configuration (just because the model needs a few samples related to a limited set of configurations in order to deliver its reliable prediction on the optimal concurrency level to be adopted).

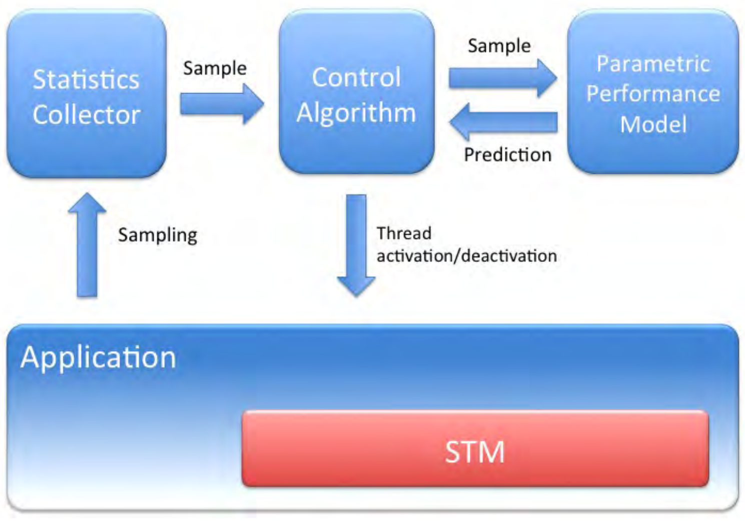


Figure 6.12: CSR-STM architecture

6.2 Concurrency Self-Regulating STM

In this section we present a concurrency self-regulating STM (CSR-STM) architecture exploiting the parametric performance model we proposed in the previous paragraphs. As for the SAC-STM platform, already presented in paragraph 5.2, SRC-STM is built using TinySTM as underlying STM layer. We also present performance study of SRC-STM, where we use the original version of TinySTM as baseline.

6.2.1 The Architecture

The architecture of the Concurrency Self-Regulating STM is depicted in Figure 6.12. A Statistic Collector (SC) provides a Control Algorithm (CA) with the average values of workload profile parameters, i.e. rs_s, ws_s, rw_a, ww_a, t_t and ntc_t, measured by observing the application on a periodic basis. Then, the CA exploits these values to calculate, through the parametric model, the transaction abort probability $p_{a,k}$ as predicted when using k concurrent threads, for each k such that $1 \leq k \leq max_{thread}$. The value max_{thread} represents the maximum amount of concurrent threads admitted for executing the application. We remark that a number of concurrent threads larger than the number of available CPU-cores typically penalizes STM performance (e.g. due to costs related to context-switches among the threads). Hence, it is generally convenient to bound max_{thread} to the maximum number of available CPU-cores. The set $\{(p_{a,k}), 1 \leq k \leq max_{thread}\}$ of predictions is used by CA to estimate the number m of concurrent threads which is expected to maximize the application throughput. Particularly, m is identified as the value of k for which

$$\frac{k}{w_{time,k} + t_{t,k} + ntc_{t,k}} \tag{6.11}$$

is maximized. In the above expression: $w_{time,k}$ is the average transaction wasted time (i.e. the average execution time spent for all the aborted runs of a transaction); $t_{t,k}$ is the average execution time of committed transaction runs; $ntc_{t,k}$ is the average execution time of ntc blocks. All these parameters refer to the scenario where the application is supposed to run with k concurrent threads.

We note that $w_{time,k} + t_{t,k} + ntc_{t,k}$ is the average execution time between commit operations of two consecutive transactions executed by the same thread when there are k active threads. Hence, expression (6.11) represents the system throughput. Now we discuss how $w_{time,k}$, $t_{t,k}$ and $ntc_{t,k}$ are estimated. We note that $w_{time,k}$ can be evaluated by multiplying the average number of aborted runs of a transaction with the average duration tr_k of an aborted transaction run when the application is executed with k concurrent threads. Thus, the average number of aborted transaction runs with k concurrent threads can be estimated as $p_{a,k}/(1 - p_{a,k})$, where $p_{a,k}$ is calculated through the presented model. So we obtain the following formula for the $w_{time,k}$ estimation:

$$w_{time,k} = \frac{p_{a,k}}{1 - p_{a,k}} \cdot tr_k \tag{6.12}$$

To calculate the average duration of an aborted transaction run (tr_k), and to estimate $t_{t,k}$ and $ntc_{t,k}$, while varying k, an hardware scalability model has to be used. In the presented version of CSR-STM, we exploited the model proposed in [75], where the function modeling hardware scalability is

$$C(k) = 1 + p \cdot (k - 1) + q \cdot k \cdot (k - 1) \tag{6.13}$$

where p and q are fitting parameters, and $C(k)$ is the scaling factor when the application runs with k concurrent threads. The values of p and q are again calculated through regression analysis. Thus, assuming that, e.g., during the last observation interval there were x concurrent threads and the measured average transaction execution time was $t_{t,x}$, CA can calculate $t_{t,k}$ for each value of k through the formula

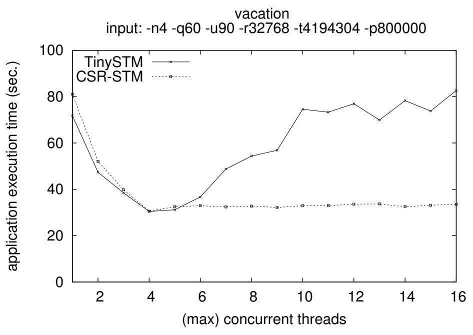
$$t_{t,k} = \frac{C(k)}{C(x)} \cdot t_{t,x} \tag{6.14}$$

The same approach is used to evaluate $ntc_{t,k}$ for each value of k, but the fitting parameters p and q are of course different than the ones used for the evaluation of $t_{t,k}$. Once estimated the number m of concurrent threads which is expected to maximize the application throughput, exactly m threads are kept active by CA during the subsequent workload sampling interval.

6.2.2 Evaluation Study

In this section we present an experimental assessment of CSR-STM, where we used vacation, kmeans and yada benchmarks, which have been run on top of the same 16-core HP ProLiant server exploited for previous experiments (As showed in Table B.1 in the Appendix B this subset of applications represent a good test case in terms of transaction length, read set and write set size, transaction execution time and contention). All the tests we present focus on the comparison of the execution time achieved by running the applications on top of CSR-STM and on top of the original version of TinySTM. Specifically, in each test, we measured, for both CSR-STM and TinySTM, the delivered application execution times while varying max_{thread} between 2 and 16. For TinySTM, max_{thread} corresponds to the (fixed) number of concurrent threads exploited by the application. While, in the case of CSR-STM, the application starts its execution with a number of concurrent threads equal to max_{thread}. However, CSR-STM may lead to changes of the number of concurrent threads setting it to any value between 1 and max_{thread}.

For each application, we calculated the values of the model parameters through regression analysis, using samples gathered observing the application running with 2 and 4 concurrent threads for the cases of vacation and yada, and including also observations with 8 concurrent threads for the case of kmeans. As for the parameters appearing in the hardware scalability model expressed in (6.13), regression analysis has been performed by using, for each application, the measured average values of the committed runs of transactions, observed with 2, 4 and 8 concurrent threads. We performed a number of runs using, for each application, different values for the input parameters. We report results achieved with two different workload profiles for each application, which are shown in



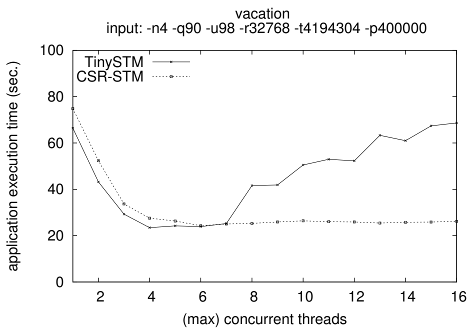


Figure 6.13: Execution time for vacation with CSR-STM and TinySTM

Figures 6.13, 6.14 and 6.15 for vacation, kmeans and yada, respectively. We explicitly report, according to the input-string syntax established by STAMP, the values of the input parameters used to run the applications. Observing the results, the advantages of CSR-STM with respect to TinySTM can be easily appreciated. For system configurations where CSR-STM is allowed to use a maximum number of threads (max_{thread}) greater than the optimal concurrency level (as identified by the peak performance delivered by TinySTM), it always tunes the concurrency level to suited values. Thus it avoids the performance loss experienced by TinySTM when making available a number of CPU-cores exceeding the optimal parallelism level. Particularly, the performance by TinySTM tends to constantly degrade while incrementing the parallelism level.

Conversely, CSR-STM prevents this performance loss, providing a performance level which is, for the majority of the cases, near to the best value, independently of the actual number of available CPU-cores for running the application. Obviously, when max_{thread} is lower than the optimum concurrency level, CSR-STM can not activate the well suited number of concurrent threads, which equals the optimal level of parallelism. Thus, for these configurations, the performance of CSR-STM is, in some cases, slightly reduced with respect

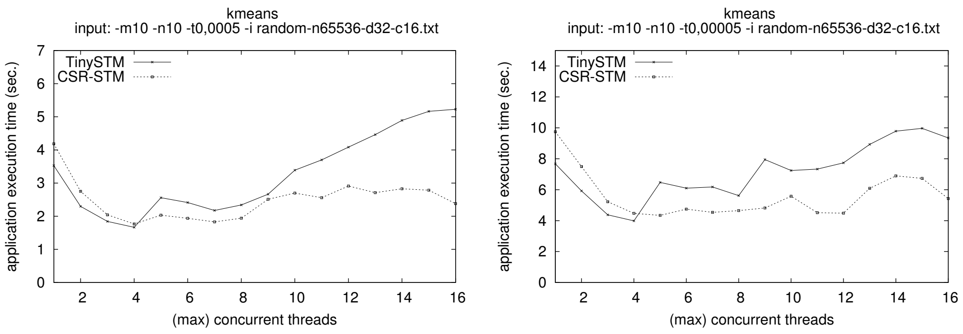


Figure 6.14: Execution time for kmeans with CSR-STM and TinySTM

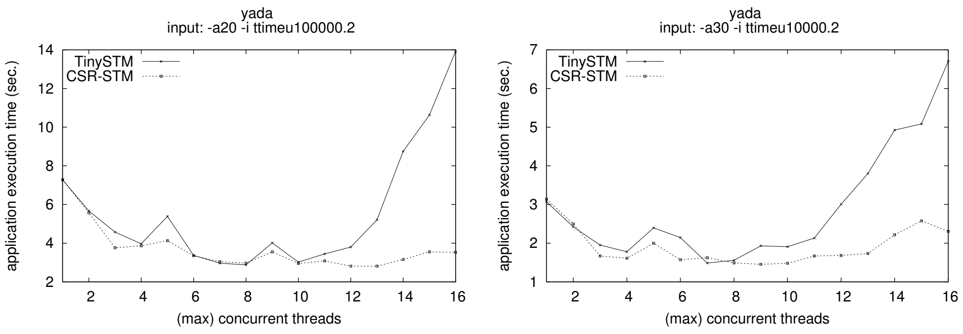


Figure 6.15: Execution time for yada with CSR-STM and TinySTM

to TinySTM due to the overhead associated with the components/tasks proper of the concurrency self-regulation mechanism. As for the latter aspect, all the components except SC (for which we measured a negligible overhead), require a single (non-CPU-bound) thread. Thus, resource demand is reduced, wrt the total application demand, of a factor bounded by $1/k$ (when k CPU-cores are available). Accordingly, the cases where CSR-STM provides lower performance than TinySTM (e.g. when max_{thread} is less than 4 for vacation and kmeans), the advantage by TinySTM progressively decreases vs max_{thread}.

Chapter 7

The Hybrid Approach

In this chapter we will describe an hybrid analytical and machine learning (AML) approach based on the combination of analytical modelling and neural networks based techniques. We combine this two approaches because in this way it is possible to address the weaknesses of one approach using the strengths of the other one and vice-versa: usually analytical models allows to obtain good extrapolation performance, that is using them makes it possible to obtain good prediction in configurations totally unknown (i.e. configuration for which it is not possible to collect performance sample). Conversely, the neural network usually doesn't have good extrapolation performance in configurations far from the ones covered by the training set, but they ensure better prediction accuracy than analytical model in configurations for which some data are available (that is data inside the training set). Combining them we obtain an hybrid approach that ensures the same accuracy of NN in configurations already explored and the same extrapolation power of analytical model in areas not already explored.

Moreover we allow the training phase required to define the "application specific" performance model to be significantly reduced, compared to pure ML techniques, while also allowing the final AML model to be significantly more precise than pure analytical approaches. In fact, it can ensure the same level of accuracy as the one provided by pure ML techniques. Further, the AML model we provide is able to cope with cases where the actual execution profile of the application, namely the workload features, can change over time, such as when the (average) size of the data-set accessed by the transactional code in read or write mode changes over time (e.g. according to a phase-behavior). This is not always allowed by pure analytical approaches [14, 16]. Overall, we provide a methodology for fast construction of a highly reliable performance model allowing the determination of the optimal level of concurrency for the specific STM-based application. This is relevant in generic contexts also including the Cloud, where the need for deploying new applications (or applications with reshuffling in their execution profile), while also promptly determining the system configurations allowing optimized resource usage, is very common.

To combine the two approaches we use the following technique: we exploit the analytical model to generate an initial training set for the neural network, that is we obtain a training set sampling the analytical model with a proper granularity. Then we execute few random application runs to collect additional real samples that will be used to update the initial training set. This update is done choosing the sample to replace using a methodology based on Euclidean distance with a constraint: if we collect a sample in which the value of the parameter k is 2, we search the sample to replace only among the samples with $k = 2$. Once defined this subset, the sample to replace is defined using the Euclidean distance calculated using the remaining parameters. This updated training set is then used to train a NN that will be used to calculate the mean wasted time predictions.

As we will show later in section 7.3.2, the hybrid approach, using the initial knowledge given by the analytical model, allows to obtain the same optimal performance prediction of neural network based approach in less time. This brings the opportunity to deploy an "optimal" system in less time. If we accept an initial phase in which the system works in suboptimal configuration, the time to deploy a working system can be furthermore reduced: it is possible to use the neural network trained with analytical model to control the system in his first interval of life. During this first phase additional real samples can be collected and can be used to update the NN training set, so the NN can be update on-line until it reaches the optimal performance.

7.1 Performance Model Aim

Typical STM oriented concurrency control algorithms [6] rely on approaches where the execution flow of a transaction never traps into operating system blocking services. Rather, they exploit spin-locks to support synchronization activities across the threads. On the other hand, several STM oriented concurrency control protocols employ on-the-fly validation schemes, actuated upon performing read-access to transactional data, which allow the early abort of the incorrectly serialized transaction (without the need for reaching its completion). In such scenarios, the primary index having an impact on the throughput achievable by the STM system (and having a reflection also in how energy is used for productive work) is the so called *transaction wasted time*, namely the amount of CPU time spent by a thread for executing transaction instances that are eventually (early) aborted.

As for the already presented approaches (chapters 5 and 6), the ability to predict the transaction wasted time, for a given application profile while varying the degree of parallelism in the execution is the fulcrum of our AML based optimization proposal. More in detail, our AML model is aimed at computing pairs of values $< w_{time,k}, k >$ where k indicates the level of concurrency and $w_{time,k}$ is the expected transaction wasted time when running with degree of concurrency equal to the value k. This values will be then used to compute the system throughput at different levels of parallelism using the Eq 5.1. Again, by exploiting Eq. 5.1, the objective of the concurrency regulation architecture we present is to identify the value of k, in the interval $[1, max\_threads]$, such that thr_k is maximized.

We will proceed along the following path. We will initially exploit a combination of the already presented approach, either analytical or machine learning, for the construction of an AML model evaluating $w_{time,k}$ for the different values of k. Essentially this will be based on introducing an algorithm for the combined usage of the two approaches. As pointed out in the introduction, such a combination will allow inheriting the best of the two worlds, namely the higher precision proper of machine learning modeling, and the reduced training time of analytical modeling, which is fundamental for fast achievement of reliable predictions.

As we will show, $w_{time,k}$ will be expressed as a function of t_t and ntc_t. However, as already showed in section 5.1.1, these quantities may depend, in their turn, on the value of k due to different thread contention dynamics on system level resources when changing the number of threads. As an example, per-thread cache efficiency may change depending on the number of STM threads operating on a given shared-cache level, thus impacting the CPU time required for a specific code block, either transactional or non-transactional. To cope with this issue, we will use the same analytical correction functions discussed in section 5.1.1 allowing, once known the value of t_t (or ntc_t) when running with k

threads, which we denote as $t_{t,k}$ and $ntc_{t,k}$ respectively, to predict the corresponding values when supposing a different number of threads. This will lead the final throughput prediction to be actuated via the Eq. 5.2. Overall, the finally achieved performance model in Eq. 5.2 has the ability to determine the expected transaction wasted time when also considering contention on system level resources (not only logical resources, namely shared-data) while varying the number of threads in the system. In fact, $w_{time,k}(t_{t,k}, ntc_{t,k}) + t_{t,k} + ntc_{t,k}$ corresponds to the predicted average execution time between the commit operations of two consecutive transactions along a same thread when there are k active threads in the system, as expressed by taking into account contention on both logical and system-level resources.

7.2 The Actual AML Model

As already done with the approach described in chapter 5 and 6, we aim at building a model for $w_{time,k}$ having the ability to capture changes in the transaction wasted time not only in relation to variations of the number of threads running the application, but also in relation to changes in the run-time behavior of transactional code blocks (such as variations of the size of amount of shared-data touched in read/write mode by the transaction). In fact, the latter type of variation may require changing the number of threads to be used in a given phase of the application execution (exhibiting a specific execution profile) in order to re-optimize performance. In chapters 5 and 6 we pointed out how capturing the combined effects of concurrency degree and execution profile on the transaction wasted time can be achieved in case $w_{time,k}$ is expressed as a function f depending on a proper set of input parameters (see section 4.2).

The objective of the AML model is to provide an approximation f_{AML} of the function f. To this purpose, we combine two different existing estimators, providing two different approximations of f. The first estimator, which we refer to as f_A is based on an analytical approach, while the second one, which we refer to as f_{ML} relies on a pure machine learning approach.

Both the two base models, namely f_A and f_{ML}, require a training phase to be actuated in order for them to be instantiated. Specifically, f_A requires the collection of some samples related to the application execution in order to compute the fitting parameters appearing in Eq.s 6.6-6.10, and to estimate tr_k (see Eq. 6.12). On the other hand, f_{ML} is constructed via the explicit reliance on a collection of a set of $(\mathbf{input}, \mathbf{output})$ training samples related to the real execution of the STM application. For both the approaches, each sample used to instantiate the model will refer to aggregate statistics (on the values of the parameters $\{rs_s, ws_s, rw_a, ww_a, t_t, ntc_t, k\}$) over multiple committed transactions, typically on the order of several thousands. However, there is a fundamental difference in the training phases to be operated for instantiating the two models.

As discussed and experimentally shown in chapter 6, the f_A model (particularly the expression for p_a) can be instantiated by relying on a (very) limited amount of run-time samples taken during real executions of the application. This implies that, upon deploying the application, a reduced number of configurations, in terms of the concurrency level (expressed by the value of the parameter k) require to be observed (and for a relatively reduce amount of time) in order to build a model having the ability to provide performance predictions in relation to very different levels of concurrency (potentially unexplored in the training phase). In other words, the f_A model offers excellent extrapolation capabilities.

This is not true for the case of f_{ML}, which typically requires to be trained via good coverage of the whole **input** domain, also in terms of the degree of concurrency k. This leads to the need for observing the application for longer time, and in differently parametrized operating modes, for reliable model instantiation. On the other hand, f_{ML} is expected to be an highly reliable estimator for f (even more reliable than f_A) in case such a good coverage of the **input** domain is guaranteed to be achieved during the training phase (see chapter 5).

We decided to combine the usage of the two modeling approaches by exploiting f_A in order to definitely shorten the length of the training phase required to instantiate f_{ML}. Overall, in our mixed modeling methodology the analytical component is used as a support to improve some aspect (namely the learning latency) of the machine learning component. This is a different way of exploiting the combination of analytical and machine learning techniques for STM optimization, compared to recent solutions like [62]. In fact, the latter ones follow the opposite path where machine learning is used to complement or to correct the analytical component.

A core aspect in our combination of analytical and machine learning models is the introduction of new type of training set for the machine learning component, which we refer to as Virtual Training Set (denoted as VTS). Particularly, VTS is a set of virtual $(\mathbf{input^v}, \mathbf{output^v})$ training samples where:

- $\mathbf{input^v}$ is the set $\{rs_s^v, rs_s^v, rw_a^v, ww_a^v, t_t^v, ntc_t^v, k^v\}$ formed by stochastically selecting the value of each individual parameter belonging to the set;
- $\mathbf{output^v}$ is the output value computed as $f_A(\mathbf{input^v})$, namely the estimation of w_{time,k^v} actuated by f_A on the basis of the stochastically selected input values.

In other words, the VTS becomes a representation of how the STM system behaves, in terms of the relation between the expected transaction wasted time and the value of configuration or behavioral parameters (such as the degree of concurrency), which is built without the need for actually sampling the real system behavior. Rather, the representation provided by VTS is built by sampling Eq. 6.12, namely f_A. We note that the latency of such sampling process is independent of the actual speed of execution of the STM application, which determines in its turn the speed according to which individual (**input**, **output**) samples, referring to real executions of the application, would be taken. Particularly, the sampling process of f_A is expected to be much faster, especially because the stochastic computation (e.g. the random computation) of any of its input parameters, which needs to be actuated at each sampling-step of f_A, is a trivial operation with negligible CPU requirements. On the other hand, the possibility to build the VTS is conditioned to the previous instantiation of the f_A model. However, as said before, this can be achieved via a very short profiling phase, requiring the collection of a few samples for the actual behavior of the STM application. Overall, we list below the algorithmic steps required for building the application specific VTS, to be used for finalizing the construction of the f_{AML} model:

Step-A. We randomly select Z different values of k in the domain $[1, max\_threads]$, and for each selected value of k we observe the application run-time behavior by taking δ real-samples, each one including the set of parameters $\{rs_s$, ws_s, rw_a, ww_a, t_t,, ntc_t, k, $tr\}$.

Step-B. Via regression we instantiate all the fitting parameters requested by Eq.s 6.6-6.10. Hence, at this stage we have an instantiation of Eq. 6.3, namely the model instance for p_a.

Step-C. We fill the instantiated model for p_a in Eq. 6.12, together with the average value of tr_k sampled in **Step-A**, and then we generate the VTS. This is done by generating δ' virtual samples $(\mathbf{input^v}, \mathbf{output^v})$ where $\mathbf{input^v} = \{rs_s^v, ws_s^v, rw_a^v, ww_a^v, t_t^v, ntc_t^v, k^v\}$ and $\mathbf{output^v} = w_{time,k^v}$ as computed by the model in Eq. 6.12. Each $\mathbf{input^v}$ sample is instantiated by randomly selecting the values of the parameters that compose it. For the parameter k the random selection is in the interval $[1, max\_threads]$, while for the other parameters the randomization needs to take into account a plausible domain, as determined by observing the actual application behavior in **Step-A**. Particularly, for each of these parameters, its randomization domain is defined by setting the lower extreme of the domain to the minimum value that was observed while sampling that same parameter in **Step-A**. On the other hand, the upper extreme for the randomization domain is calculated as the value guaranteeing the 90-percentile coverage of the whole set of values sampled for that parameter in **Step-A**, which is done in order to reduce the effects due to spike values.

After having generated the VTS in **Step-C**, we use it in order to train the machine learning component f_{ML} of the modelling approach. However, training f_{ML} by only relying on VTS would give rise to a final f_{ML} estimator identical to f_A given that the curve learned by f_{ML} would exactly correspond to the one modelled by f_A. Hence, in order to improve the quality of the machine learning based estimator, our combination of analytical an machine learning methods relies on additional algorithmic steps where we use VTS as the base for the construction of an additional training set called Virtual-Real Mixed Training Set (denoted as VRMTS). This set represents a variation of VTS where some virtual samples are replaced with real samples taken by observing the real behavior of the STM application, still for a relatively limited amount of time. More in detail, the following two additional algorithmic steps are used for constructing the VRMTS:

Step-D. We select Z' different values for k (in the interval $[1, max\_threads]$), and for each selected value we observe the application run-time behavior by taking δ'' real training samples $(\mathbf{input^r}, \mathbf{output^r})$.

Step-E. We initially set VRMTS equal to VTS. Then we generate the final VRMTS image via an iterative procedure where we substitute at each iteration one element in VRMTS with one individual $(\mathbf{input^r}, \mathbf{output^r})$ sample from the sequence of samples taken in **Step-D**, until this sequence ends.

The rationale behind the construction of VRMTS is to improve the quality of the final training set to be used to build the machine learning model by complementing the virtual samples originally appearing in VTS with real data related to the execution of the application. Two things need to be considered in this process: (1) the actual length of **Step-D** could be further reduced by reusing (all or part of the) real samples of the application execution taken in **Step-A**, which were exploited in **Step-B** for computing the fitting parameters for the f_A model; (2) the substitution in **Step-E** could be actuated according to differentiated policies.

As for the latter aspect, we have decided to use a policy based on Euclidean distance, in order to avoid clustering phenomena leading the final VRMTS image to containing training samples whose distribution within the whole domain significantly differs from the original distribution determined by the random selection process used in **Step-C** for the construction of VTS. More in detail, the victim selection policy we have adopted to replace iteratively any sample while generating the final VRMTS works as follows:

- given a collected real sample of the application execution $s^r = (rs_s^r, ws_s^r, rw_a^r, ww_a^r, t_t^r, ntc_t^r, k^r)$ the subset $S_{k^r} = \{(rs_s, ws_s, rw_a, ww_a, t_t, ntc_t, k) | k = k^r\}$

of VRMTS is computed. Actually, S_{k^r} is the subset of samples for which the level of parallelism k they refer to is the same as the level of parallelism characterizing the real sample to be used for replacement in the current iterative step;

- the actual sample in VRMTS to be replaced with s^r is identified inside the subset S_{k^r} using the Euclidean distance as computed on all the parameters characterizing the sample except k (namely rs_s, ws_s, rw_a, ww_a, t_t and ntc_t). Particularly, the victim is the sample s^* belonging to S_{k^r} which is closest to s^r.

We note that the above Euclidean distance based policy may lead in intermediate steps to evict from VRMTS some previously inserted real sample. This may happen in case the closest sample to the one currently being inserted in VRMTS is a real sample (which was inserted in a previous iteration). This is not a drawback of our victim selection policy, rather it is the reflection of the fact that we prevent clustering effects of the elements included in the final image of VRMTS, which may lead some portions of the domain not to be sufficiently represented within the set.

Once achieved the final VRMTS image, we use it to train f_{ML} in order to determine the final AML model. Overall, f_{AML} is defined as the instance of f_{ML} trained via VRMTS.

7.2.1 Correcting Factors

As pointed out, the instantiation of the f_{AML} model for the prediction of $w_{time,k}$ needs to be complemented with a predictor of how t_t and ntc_t are expected to vary vs the degree of parallelism k. In fact, $w_{time,k}$, as expressed by the instance of machine learning predictor trained via VRMTS depends on t_t and

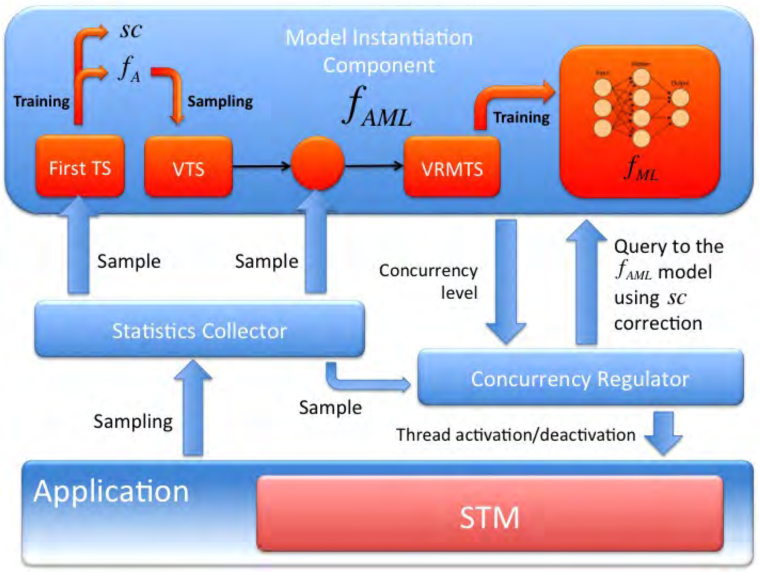


Figure 7.1: System architecture

ntc_t. Moreover, the final equation establishing the system throughput, namely Eq. 5.2, which is used for evaluating the optimal concurrency level, also relies on the ability to determine how t_t and ntc_t change when changing the level of parallelism (due to contention on hardware resources). As already done with the Neural Network based approach presented in chapter 5, to cope with this issue we decided to complement the whole process with the instantiation of correcting functions aimed at determining (predicting) the values $t_{t,k}$ and $ntc_{t,k}$ once know the values of these same parameters when running with parallelism level $i \neq k$. To achieve this goal, the samples taken in **Step-A** of the above presented process are used to build, via regression, the function expressing the variation of the number of clock-cycles that the CPU-core spends waiting for data or instructions to come-in from the RAM storage system. We recall that the collection of training samples in **Step-A** should be made very short, hence referring to a limited number of values of the concurrency level k. However, the expectation is that the number of clock-cycles spent in waiting phases should scale (almost) linearly vs the number of concurrent threads used for running the application. Hence, regression on a limited number of samples should suffice for reliable instantiation of the correction functions. A detailed description of the functions used for the correction of the parameters t_t and ntc_t can be found in section 5.1.1

7.3 Experimental Evaluation

7.3.1 The AML Based Concurrency Regulation Architecture

We have implemented a fully featured STM concurrency regulation architecture based on AML, which we refer to as AML-STM, whose organization is presented in Figure 7.1. The core STM layer exploited in our implementation is again the open source middleware TinySTM. AML-STM is made up by three building blocks, namely:

- a Statistics Collector (SC);
- a Model Instantiation Component (MIC); and
- a Concurrency Regulator (CR).

The MIC module initially interacts with CR in order to induce variations of the number of running-threads i so that the SC module is allowed to perform the sampling process requested to support **Step-A** of the instantiation of the AML model. After the initial sampling phase, the MIC module instantiates f_A (and the correction function sc) and computes VTS. It then interacts again with CR in order to induce variations of the concurrency level k that are requested to support the sampling process (still actuated via SC) used for building VRMTS (see **Step-D** and **Step-E**). It then instantiates f_{AML} by relying on a neural network implementation of the f_{ML} predictor, which is trained via VRMTS. Once the f_{AML} model is built, MIC continues to gather statistical data from SC, and depending on the values of $w_{time,k}$ that are predicted by f_{AML} (as a function of the average values of the sampled parameters rs_s, ws_s, rw_a, ww_a, $t_{t,k}$, and $ntc_{t,k}$), it determines the value of k providing the optimal throughput by relying on Eq. 5.2. This value is filled in input to CR (via queries by CR to MIC), which in its turn switches off or activates threads depending on whether the level of concurrency needs to be decreased or increased for the next observation period.

We note that the length of the phases requested for eventually instantiating f_{AML} depends on the amount of samples that are planned to be taken in **Step-A** and in **Step-D** of the model construction (see the parameters Z, δ, Z' and δ'' in the detailed description of these steps). We will evaluate the effectiveness of our AML modeling approach, and compare this approach with pure analytical or machine learning based methods, while varying the length of these sampling phases. We note that the shorter such a length, the more promptly the final performance model to be used for concurrency regulation is available. Hence, reduction of the length of these phase, while still guaranteing accuracy of the finally built performance model, will allow more prompt optimization of the run-time behavior of the STM based application. As hinted, this is relevant in scenarios where applications are dynamically deployed, and need to be promptly optimized in terms of their run-time behavior in order to improve the fruitful usage of resources and to also improve the system energy efficiency (via reduction of wasted CPU time), such as when applications are hosted by PaaS providers on top of STM-based platforms.

7.3.2 Experimental Data

The data we report in this section refer to the execution of four applications belonging to the STAMP benchmark suite: intruder, kmeans, yada, vacation. As showed in Table B.1 (see Appendix B) this subset of applications represent a complete test case in terms of transaction length, read set and write set size, transaction execution time and contention. So it should represent a good and complete testbed to prove the effectiveness of our approach. These applications have been run on top of the aforementioned 16-core HP ProLiant machine. This section is divided in two parts. In the first one we provide an experimental support of the feasibility of our AML approach. Specifically, we provide data related to how the prediction error of $w_{time,k}$ changes over time (namely vs the length of the sampling phase used to gather data to instantiate the performance model) when comparatively considering our AML model and the two base models, pure analytical and pure machine learning, exploited for building AML.

Successively, we provide experimental data related to how the concurrency regulation architecture based on AML, namely AML-STM, allows more prompt achievement of optimized run-time performance and optimized energy usage, when compared to the concurrency regulation architectures we have presented in chapters 5 and 6, where concurrency regulation takes place by exclusively relying on an analytical performance model or on a pure machine learning approach. We will refer to these two architectures as CSR-STM and SAC-STM, respectively. We note that both these architectures have been implemented by relying on TinySTM as the core STM layer, hence our study provides a fair comparison of the different performance modeling and optimization approaches, when considering the same STM technology and implementation. Also, we feel that comparing our AML approach with literature approaches addressing the very same problem (namely the dynamic selection of the optimal value of the number of threads in scenarios where the application execution profile can change over time) is the more reliable way of assessing the present proposal ([1]). In fact, comparing AML with approaches based on different rationales (like the ones based on transaction scheduling, see [71]) would lead to compare solutions that can be integrated and make synergy, thus not representing alternatives excluding each other. Overall, in this chapter our study is oriented to the evaluation of real mutual-excluding alternatives, specifically targeting the problem of regulating the level of concurrency. However, to provide a complete overview of the performance reachable using different approaches, in Chapter 8 we will show the results of a study aimed at comparing the performance ensured by our concurrency regulation approaches and by other orthogonal performance optimization approaches.

Part A - Model Accuracy

Table 7.1 shows the average error rate and the error rate variance of wasted time predictions, for analytical model and Neural Network based approaches. These values are obtained training the analytical model and the neural network with the same training set. This training set contains enough samples to allows the neural network to obtain better prediction performance than analytical model. The values showed by the table are a proof that if enough samples are available the Neural Network is preferable to analytical model approach.

To evaluate the generalization capacity of the three approaches we train all the predictor starting from the same training set S. It is a small training set

[1]The proposals in, e.g. [15, 14, 16], are suited for selecting and/or regulating concurrency with static execution profiles, where, e.g., read and write set size does not change over time. We exclude therefore these solutions in our comparative study.

yada		
	mean error	**error variance**
Analytical model	0,0419930428	0,0008199737
Neural Network	0,0115209801	0,0002040777

vacation		
	mean error	**error variance**
Analytical model	0,0362778191	0,0032975101
Neural Network	0,030409852	0,0032735876

kmeans		
	mean error	**error variance**
Analytical model	0,0477515107	0,0015677014
Neural Network	0,0273179969	0,000881993

intruder		
	mean error	**error variance**
Analytical model	0,0869683006	0,0041785981
Neural Network	0,0362397331	0,0021788312

Table 7.1: Average error and variance comparison

that contains only samples related to levels of parallelism equals to 2 and 4. We use S to train the NN and the analytical model. For the training of the predictor based on the mixed approach we build a new training set S_m merging S with additional samples obtained sampling the trained analytical model. In this way we obtain an initial hybrid training set that contains some real data and some extrapolated data.

In figures 7.2 and 7.3 we can see the results that can be obtained with the three different types of predictor. The graphs in 7.2 show normalized wasted time predictions vs normalized real wasted time values for yada benchmark. In the first graph, that shows neural network prediction vs real wasted time values, the points are scattered. It implies that the predicted values are very different respect to real wasted time values, this is due to the small size of the training set that doesn't allow the neural network to enrich good prediction performance. The second graph shows the model predictions vs real time values. As we can see the points are concentrated in a limited area around the bisector line and this is a proof that the predictions of the analytical model are better than the ones provided by the neural network, that is the model has better generalization/extrapolation capacity and the error is limited. The third graph shows the mixed approach predictions vs the real wasted time values. In this case the points are concentrated in an smaller area around the bisector line. It means that the mixed approach allows to obtain better predictions than analytical model. This is due to the capability of the mixed approach of exploit both the strengths of the first two approaches: generalization capacity of analytical model and accuracy of neural network. The graphs in 7.3 show normalized wasted time prediction vs normalized real wasted time values for kmeans benchmark. In the first graph, that shows neural network predictions vs real wasted time values, we

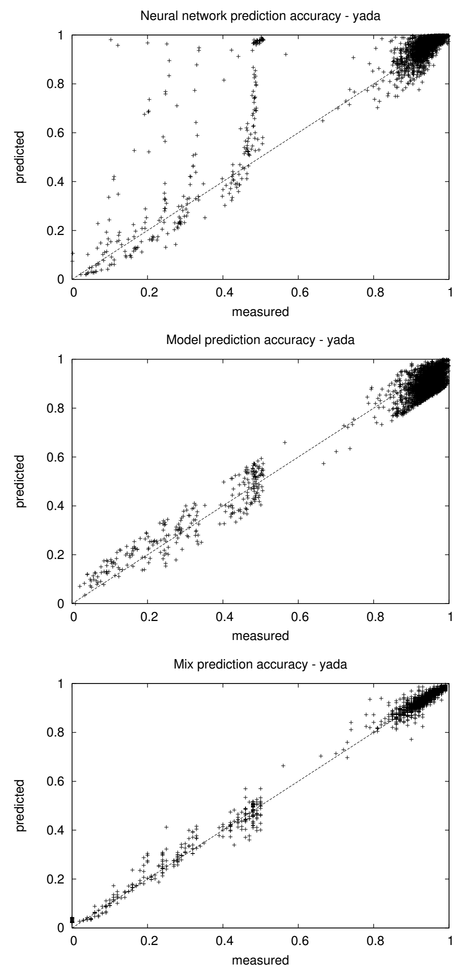


Figure 7.2: Predictions accuracy - yada benchmark

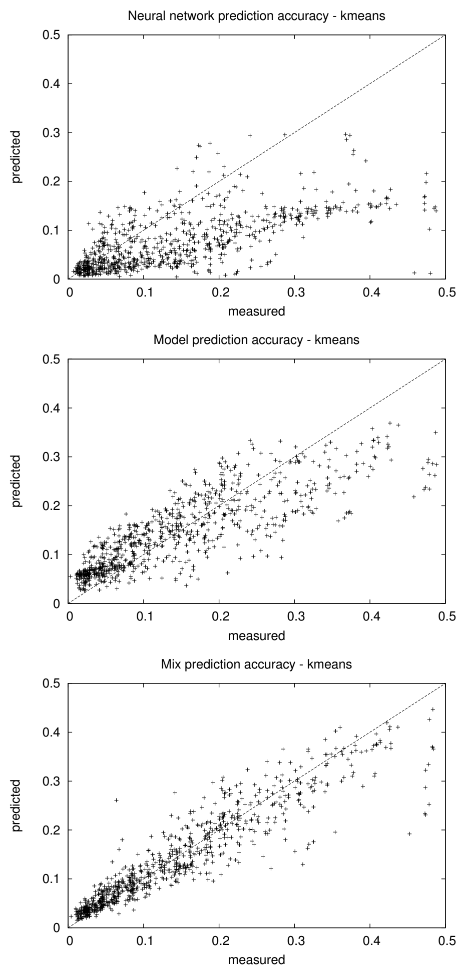


Figure 7.3: Predictions accuracy - kmeans benchmark

can see that the points are scattered and far from the bisector line. It implies that the predicted values are very different respect to the real wasted time values and most of the time the NN underestimate the mean wasted time. In this case too, the small size of the training set doesn't allow the neural network to enrich good prediction performance. In the second graph we can see the model prediction vs real time values. In this case the points are still scattered, but less respect to NN graph. Moreover they are distributed around the bisector line. It implies that the model ensures better prediction performance that pure NN approach (with the available training samples). The third graph shows the mixed approach predictions vs the real wasted time values. As we can see the points are concentrated in an smaller area around the bisector line. It means that the mixed approach allows to obtain better predictions than analytical model. In this case too, it is due to the capacity of the mixed approach of exploit both the strengths of neural networks and analytical modelling approaches.

To determine how the estimation accuracy of $w_{time,k}$ provided by the AML approach varies vs the length of the sampling phase used to gather profiling data on top of which the performance model is built, and to compare such accuracy with the one provided by pure analytical (f_A) or pure machine learning (f_{ML} trained on real samples) methods, we have performed the following experiments. We have profiled STAMP applications by running them with different levels of concurrency, which has been varied between 1 and the maximum amount of available CPU-cores, namely 16. All the samples collected up to a point in time have been used either to instantiate f_A via regression, or to train f_{ML} in the pure machine learning approach. On the other hand, for the case of f_{AML} they are used according to the following rule. The 10% of the initially taken samples in the observation interval are used to instantiate f_A (see **Step-A** and

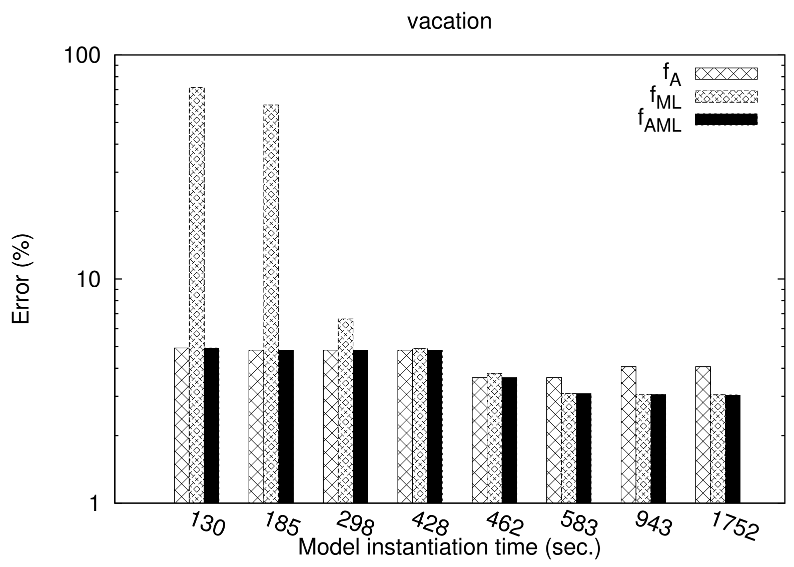


Figure 7.4: Prediction error comparison - vacation

Step-B in Section 7.2), which is then used to build VTS, while the remaining 90% are used to derive VRMTS (see **Step-D** and **Step-E** in Section 7.2). In this scheme the cardinality of the VTS, from which VRMTS is build, has been fixed at 1500 elements. Also, each real sample taken during the execution of the application aggregates the statistics related to 4000 committed transactions, and the samples are taken in all the scenarios along a single thread, thus leading to similar rate of production of profiling data independently of the actual level of concurrency while running the application. Hence, the knowledge base on top of which the models are instantiated is populated with similar rates in all the scenarios.

Then for different lengths of the initial sampling phase (namely for different amounts of samples coming from the real execution of the application), we instantiated the three different models and compared the errors they provide in predicting $w_{time,k}$. These error values are reported in Figures 7.4-7.7, and refer

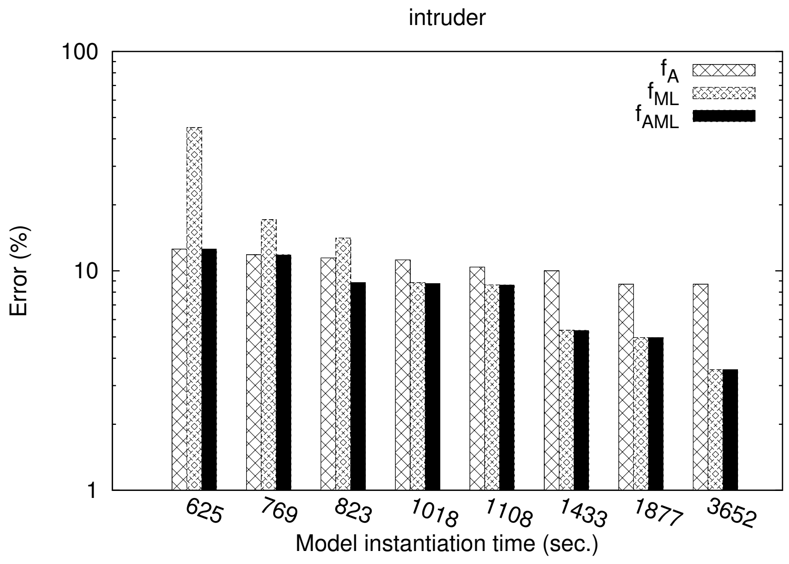


Figure 7.5: Prediction error comparison - intruder

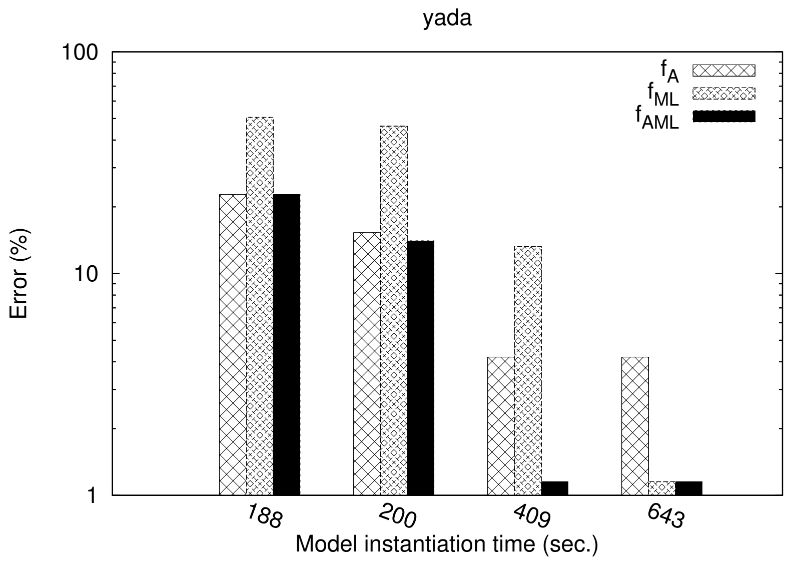


Figure 7.6: Prediction error comparison - yada

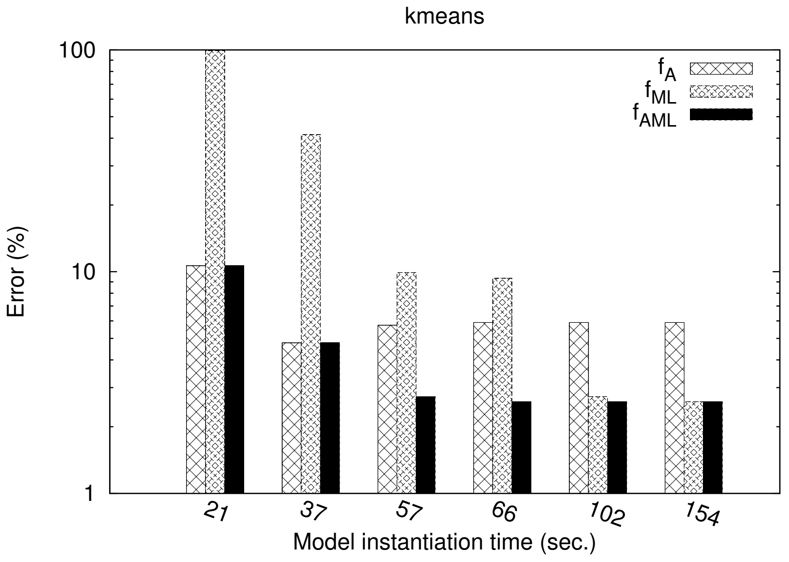


Figure 7.7: Prediction error comparison - kmeans

to the average error in the prediction while comparing predicted values with real execution values achieved while varying the number of threads running STAMP applications in between 1 and the maximum value 16. Hence, they are average values over the different possible configurations of the concurrency degree for which predictions are carried out. Also, we have normalized the number of real samples used in each approach in such a way that the x-axis expresses the actual latency for model instantiation (not only for real samples collection), hence including the latency (namely the overhead) for VTS and VRMTS generation and actual training of f_{ML} over VRMTS in case of the AML approach. This allowed us to compare the accuracy of the different models when considering the same identical amount of time for instantiating them (since for models requiring more processing activities in order for them to be instantiated, we recover that time by reducing the actual observation interval, and hence the number of real samples provided for model construction).

By the data we can see how the AML modeling approach always provides the minimal error independently of the length of the application profiling phase. Also, with the exception of *vacation* and *intruder*, AML allows achieving minimal errors (on the order of 2-3%) in about half of the time requested by the best of the other two models for achieving the same level of precision. On the other hand, for *intruder*, AML significantly outperforms the other two prediction models for different lengths of the application sampling period. As for *vacation*, AML provides close-to asymptotically minimal prediction error even with a very reduced amount of available profiling samples. These data support the claim of high accuracy of the predictions by AML, guaranteed via very reduced time for instantiating the application specific performance model.

Part B - Performance and Energy Efficiency

To demonstrate the effectiveness of AML in allowing prompt deliver of optimized performance (and prompt improvement of energy usage), once instantiated the performance models at some point in time according to the settings presented in Section 7.3.2, we evaluated both: (A) the transaction throughput, given that the concurrency level is dynamically regulated according to the predictions by the instantiated model and (B) the average energy consumption (joule) per committed transaction. For both the parameters, we also report the values achieved by running the application sequentially on top of a single thread and fixing the number of threads to the maximum value of 16 (we refer to this configuration as TinySTM in the plots), which allows us to establish baseline values for the assessment of both speedups and energy usage variations by the runs where the degree of concurrency is dynamically changed on the basis of the performance model predictions.

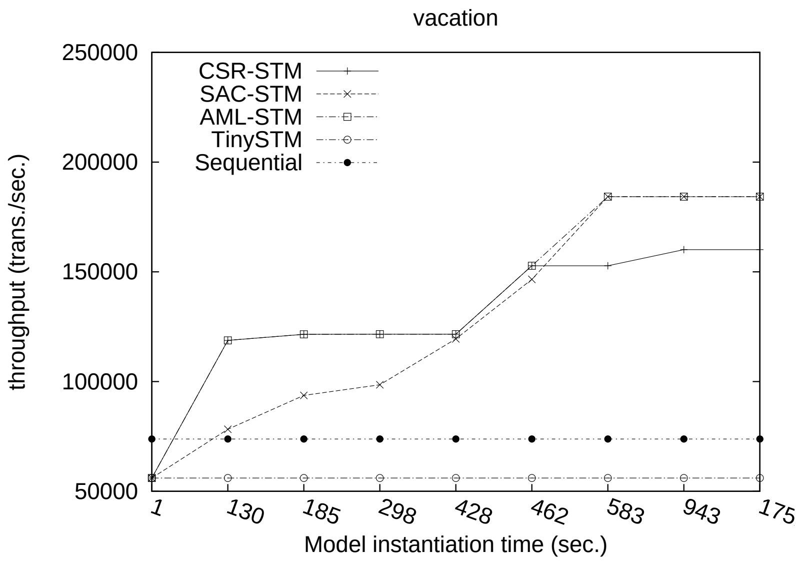


Figure 7.8: Throughput - vacation

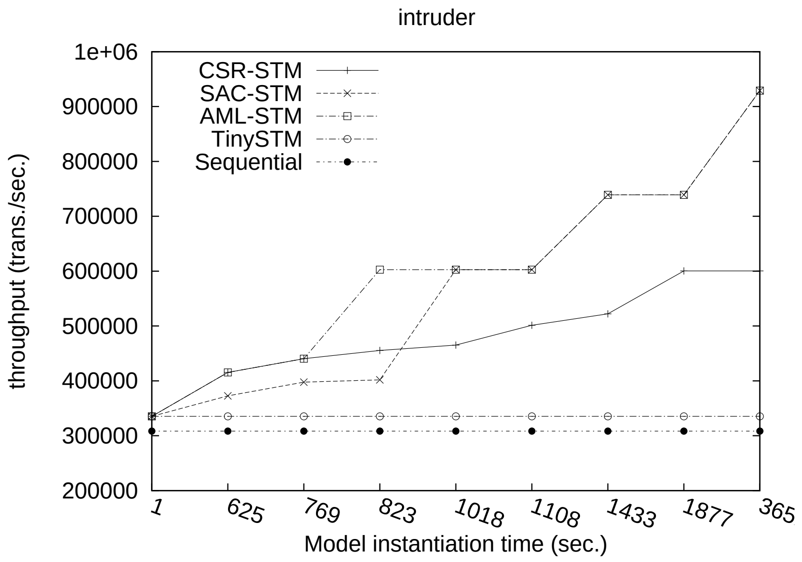


Figure 7.9: Throughput - intruder

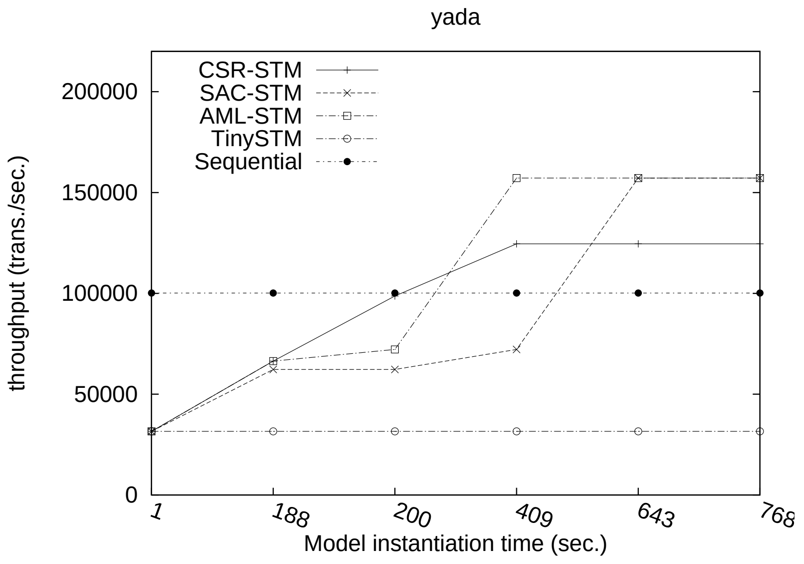


Figure 7.10: Throughput - yada

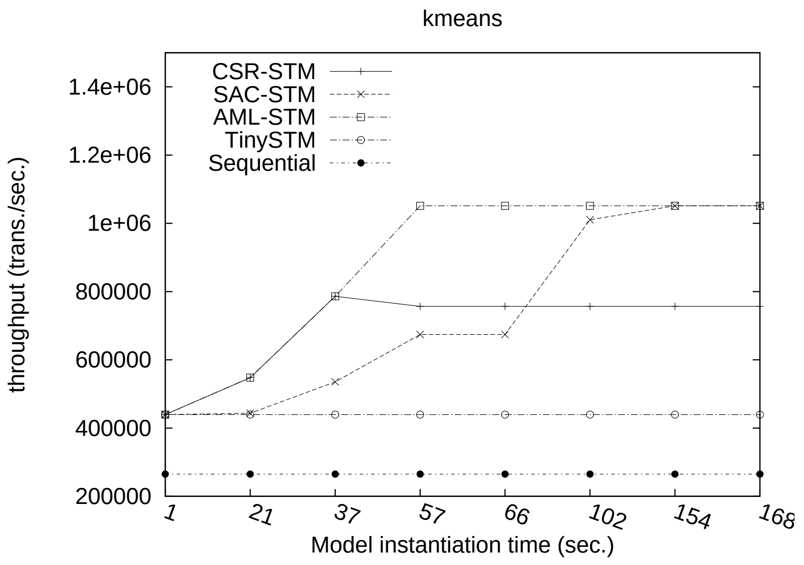


Figure 7.11: Throughput - kmeans

By the throughput data in Figures 7.8-7.11, we see how dynamic concurrency regulation based on AML allows the achievement of improved or even the peak observable throughput values much earlier in time, when compared to what happen with the pure analytical and the pure machine learning approaches. Also, the pure analytical approach is typically not able to provide the peak observed throughput, independently of the length of the sampling period during which the knowledge base for instantiating the model is being constructed. Also, for some benchmark, such as *kmeans*, the time requested by the pure machine learning based approach in order to instantiate a model guaranteing the peak observed performance is one order of magnitude longer than what required for the instantiation of the AML model. For other benchmarks, such as *yada*, the AML approach requires on the order of 40% less model-instantiation time to achieve a model providing the peak performance. We also note that for most of the benchmarks, the TinySTM configuration where all the available 16 CPU-cores are used to run a fixed number of 16 concurrent threads, typically leads to a speed-down wrt the sequential run. This indicates how the execution profiles for the STAMP applications are not prone to exploitation of uncontrolled concurrency levels, which leads the actually observed speedup values, promptly achievable via AML, to be representative of a significant performance boost.

As for data related to energy efficiency, reported in Figures 7.12-7.15, we see how both the pure analytical and the AML approaches allow prompt achievement of reduction of the energy requested per transaction commit. This is not guaranteed by the pure machine learning approach. Also, the AML approach allows the achievement of optimized tradeoffs between execution speed and energy consumption. In fact, even though the pure analytical approach allows reducing the energy consumption for the *yada* benchmark when considering longer

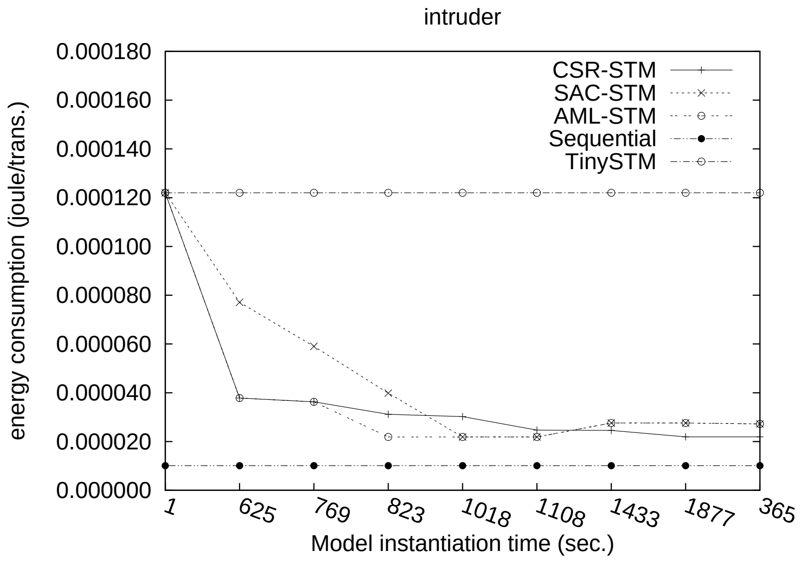


Figure 7.12: Energy consumption per committed transaction - intruder

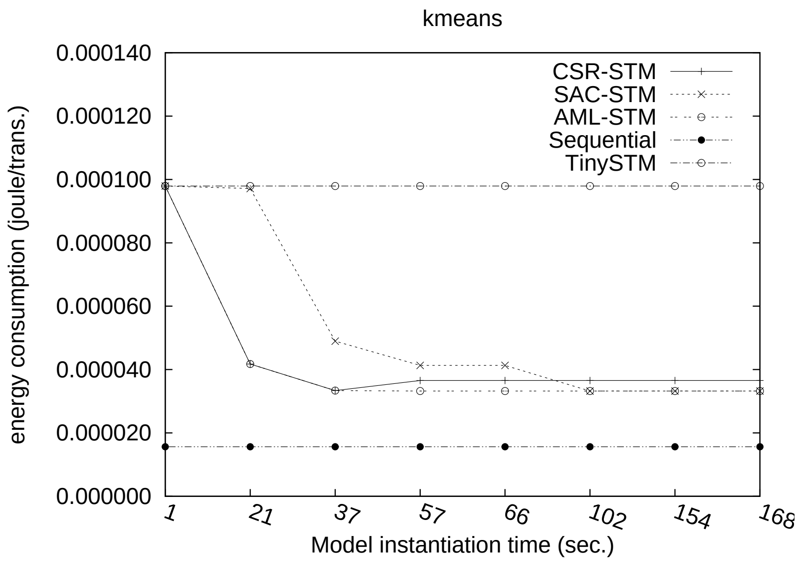


Figure 7.13: Energy consumption per committed transaction - kmeans

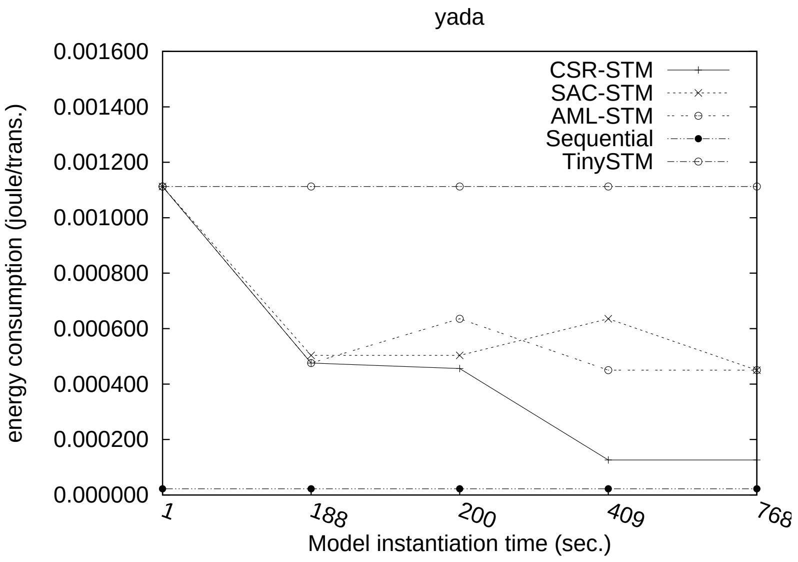


Figure 7.14: Energy consumption per committed transaction - yada

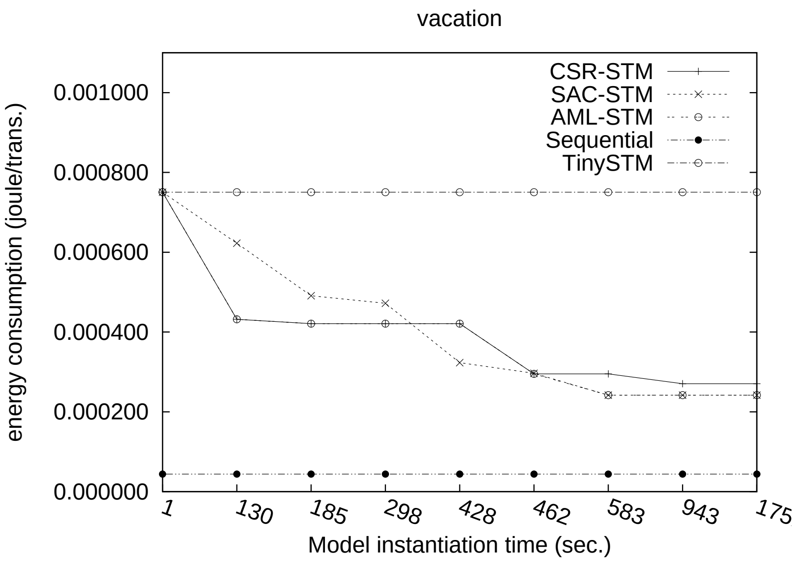


Figure 7.15: Energy consumption per committed transaction - vacation

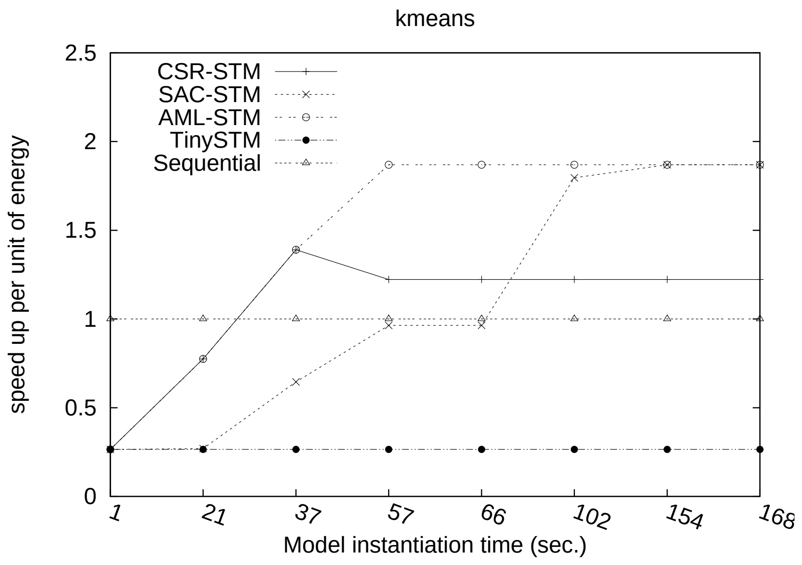


Figure 7.16: Iso-energy speedup curves

time for model instantiation, this is achieved by clearly penalizing the achieved throughput.

To provide more insights into the relation between speed and usage of energy, we report in Figure 7.16 the curves showing the variation of the ratio between the speedup provided by any specific configuration (again while varying the performance model instantiation time) and the energy scaling per committed transaction (namely the ratio between the energy used in a given configuration and the one used in the sequential run of the application). As an example we report these curves for the *kmeans* benchmark, however the corresponding curves for the other benchmark applications could be derived by combining the previously presented curves. Essentially, the curves in Figure 7.16 express the speedup per unit of energy, when considering that the unit of energy for committing a transaction is the one employed by the sequential run. Hence they express a kind of iso-energy speedup. Clearly, for sequential runs this curve has constant value equal to 1. By the data we see how the AML approach achieves the peak observed iso-energy speedup for a significant reduction of the performance model instantiation time. On the other hand, the pure analytical approach does not achieve such a peak value even in case of significantly stretched application sampling phases, used to build the model knowledge-base. Also, the configuration with concurrency degree set to 16, namely TinySTM, further shows how not relying on smart and promptly optimized concurrency regulation, as the one provided by AML, degrades both performance and energy efficiency.

Chapter 8

STMs Performance Comparison

In this chapter we provide the results of a study that compares the performance of the approaches presented in this thesis with those of other approaches aimed to optimize the performance of STM based applications. We don't include in the study the AML approach presented in Chapter 7 because, in terms of obtainable performance, it is equivalent to the pure NN based approach presented in Chapter 5 and we don't include the analytical model based approach presented in Chapter 6 because in some cases it doesn't reach the same prediction accuracy ensured by the NN based approach. For the comparison we have selected a set of standard and self-adaptive STM middlewares:

- TinySTM, used as baseline configuration for the evaluation;
- SwissTM [35] that, compared to TinySTM, provide a different concurrency control mechanism.
- SAC-STM, ATS-STM [12] and Shrink [13] as self-adaptive STM middlewares. They are used to show how self-adapting computation can capture

the actual degree of parallelism and/or logical contention on shared data in a better way respect to not adaptive implementations, enhancing even more the intrinsic benefits provided by software transactional memory;

We executed all the STAMP benchmark suite applications on top of each STM implementation using the platform described in section 2.2. The obtained results will be discussed in Section 8.2.

8.1 Reference Architectures and Configurations

We hereby provide a brief recall of the approaches used to carry out the evaluation study. For each approach, we give informations about the configuration used for our experiments.

8.1.1 SwissTM

SwissTM is a lock based STM that uses an encounter-time (pessimistic) conflict detection mechanism for conflicts between concurrent writes and a commit-time (optimistic) conflict detection mechanism for conflicts between concurrent read and write. It uses a two-phase contention manager with random linear back-off that is able to ensure the progress of long transactions while inducing no overhead on short ones. The API of SwissTM is word-based, that is it enables transactional access to arbitrary memory words. It guarantees opacity [25], a property similar to serializability in database systems [76] (the main difference is that all transactions always observe consistent states of the system). This implies that transactions cannot, e.g., use stale values, and that they do not require periodic validation or sand-boxing to prevent infinite loops or crashes due to accesses to inconsistent memory states. SwissTM does not provide any guarantees for the code that accesses the same data from both inside and outside of transactions and it is not privatization safe [77]. This could make programming with this STM slightly more difficult in certain cases.

8.1.2 TinySTM

The basic setup of TinySTM implements the Encounter-Time Locking (ETL) algorithm. It is an algorithm based essentially on locks, which are acquired (on a per-word basis) whenever a write operation is to be performed on a shared variable. In particular, TinySTM relies on a shared array of locks, where each lock is associated with a portion of the (shared) address space. Upon a write operation, the transaction identifies which lock is covering the memory region which will be affected by the write, and atomically reads its value. A lock value is composed by an integer number, the least significant bit of which tells whether the lock is currently owned by a running transaction. The remaining bits specify the current version number associated with that particular memory region. The writing transaction, therefore, reads the lock bit and determines whether that memory region is already owned or not. In the positive case, the transaction checks whether the lock its owned by the transaction itself. If it is so, the new value is directly written, otherwise for a specified amount of time the transaction gets into a waiting state. In the negative case, an atomic compare-and-swap (`CAS`) operation is used to try to acquire the lock. A failure in the `CAS` operation indicates that the lock has been (concurrently) acquired by another transaction, so the execution of the write operation falls into the first aforementioned case. The current version number stored in the lock is used upon read operations, to check whether the memory region has been updated by other transactions. In particular, before reading, the transaction checks whether the lock is owned, then reads the counter value, then performs the read operation and reads the counter value again. If between the two counter reads its value is not changed, then the read value can be regarded as consistent. We have used TinySTM relying on the *write-back* scheme for the propagation of memory updates. It buffers all the updates in a *write log*, which, upon commit operation, is flushed to memory. This approach reduces the abort time and simplifies guaranteeing consistency of read operations. ETL has two main advantages [7]: On the one hand, by detecting conflicts early it can provide a transaction throughput increase, reducing the amount of wasted work executed by transactions. On the other hand, read-after-writes can be handled efficiently, providing a non-negligible benefit whenever write sets are large enough.

8.1.3 TinySTM Adaptive Configurations

All the adaptive STM configurations included in the comparison are obtained improving the standard version of TinySTM with some features aimed to optimize the application performance. We provide below a short description of the improvement introduced by each implementation.

SAC-STM

This STM implementation, described in detail in Section 5.2, exploits a machine-learning based controller which regulates the amount of active concurrent threads along the execution of the application. Specifically, a neural network is trained in advanced to learn existing relations between the average wasted transaction execution time (i.e. the average time spent by a thread executing aborted transactions for each committed transaction) and: (i) a set of parameters representing the current workload profile of the application, and (ii) the number of active concurrent threads. The neural network is exploited by the controller to estimate the optimal number of concurrent threads to be kept active. This is done by evaluating the expected throughput as a function of the application's current workload and of the number of active threads k, with $1 \leq k \leq max_{thread}$, where max_{thread} is a system configuration parameter denoting the maximum number of concurrent threads which can be activated. The controller also exploits a model to predict the system's hardware scalability of the time spent in transactions and of the time spent in non-transactional code as a function of the number of active concurrent threads. At the end of each workload sampling interval, during which all workload profile parameters are evaluated on the basis of statistics collected through runtime measurements, the controller performs new throughput estimations and then activates only a certain number of threads so that throughput is expected to be higher.

ATS-STM

This STM implementation has been described in detail in Section 3.1. It is based on Adaptive Transaction Scheduling (ATS), a transaction-scheduling algorithm relying on runtime measurement of the Contention Intensity (CI), an index that gives an estimation of the contention that a transaction encounters during its execution. Each thread maintains its own contention intensity. Before starting a new transaction, if the current value of CI exceeds a given threshold, then the thread stalls and the transaction is inserted within a queue shared by all threads. Otherwise, the thread immediately execute the transaction. Transactions stored inside the queue are serialized and executed according to the FIFO order.

Shrink

This STM implementation has been described in detail in Section 3.1, too. It uses a transaction-scheduling algorithm based on temporal locality, i.e. on the fact that consecutive transactions in a thread access the same data objects. Similarly to ATS-STM, in Shrink the scheduler is activated if the transaction success rate is below a given threshold. Yet, rather than serializing all transactions, a contention probability is evaluated on the read- and write-sets. Specifically, before starting a new transaction, an estimation of the probability of write contention among the entries in the predicted read-/write-sets is computed. This is done by checking if there is an intersection between the predicted write-set of (predicted) concurrent transactions and the union of the read-set and the write-set of the transaction to be executed. In the positive case, the new transaction must be serialized. The predicted read-set of starting transactions is the union of the read-sets of the last n executed transactions, where n is a configuration parameter called *locality window*. The predicted write-set of a transaction includes all the data objects written by the same transaction during previous executions (if any, otherwise the write-set in assumed to be empty).

8.2 Experimental Data

In this sections we discuss the performance reachable with the just described STM implementations. To cover a wide range of transactional workload profiles we executed each application using different input parameters, chosen following the methodology described in section 5.3.1. For each application we will show four graphs, one for each tested configuration. In the graphs we report the application execution time varying the number of maximum available threads.

This means that the STM implementations with concurrency regulation capability will start their execution with a number of thread equal to the maximum available one and than they will regulate that number on the basis of the application workload, instead the other implementations run with the maximum number of available threads for all the application lifetime. Comparing the results obtained with TinySTM and SAC-STM that we will show in the current section with the ones already showed for the same STM frameworks in Section 5.3.3 we can see that the latter are always worst. This is due to the usage of a different contention manager: in the experiments presented in Section 5.3.3 we used the contention manager CM_MODULAR (it allows to dynamically choose contention manager between a predefined set and it may be a bit slower than other CMs because of its flexibility), in the ones we will present in the current chapter we used instead the contention manager CM_SUICIDE (when a conflict occurs, it kills the transaction that detects the conflict). The results obtained in both the configurations show that our concurrency regulation approaches, regardless of the used contention manager, are able to choose always the optimal concurrency level further proving their generality and effectiveness.

In Figure 8.1 we present the results for the intruder benchmark. As we can see, for all considered configurations, the application execution time achieved with TinySTM decreases when increasing the number of used threads up to 4-8, while for greater values it drastically increases. Similar results are obtained with ShrinkSTM, but this implementation always reaches worst performance than TinySTM (except between 14-16 threads for the first and the forth configurations). About ATS-STM we can see that it ensures the same performance of the other implementations for concurrency level between 1 and 4. With higher level it isn't able to improve its performance but it is able to maintain fairly

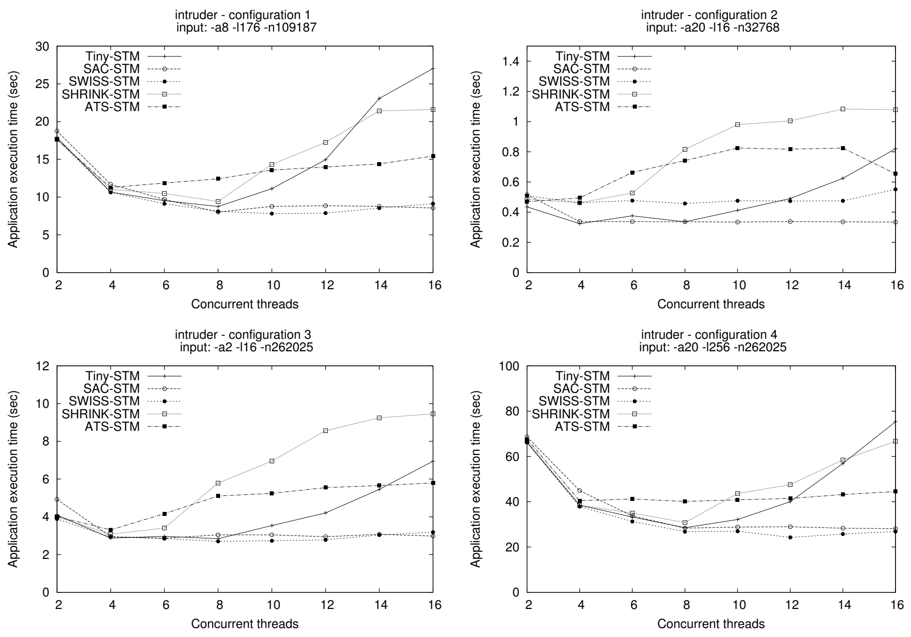


Figure 8.1: Application execution time - intruder benchmark

constant the application execution time. For all the tests, SAC-STM achieves very good results independently of the maximum amount of allowed concurrent threads. In fact, in scenarios where max_{thread} is less than the amount of threads giving rise to the optimum case for TinySTM, the results achieved with SAC-STM and the other STM are comparable. With more threads, SAC-STM is able to constantly ensure an application execution time very near to the best one achieved with TinySTM. About SwissTM we can see that it is able to ensure the same optimal performance of SAC-STM for *configuration 3* and *configuration 4*. Only for *configuration 2* SwissTM constantly reaches worst performance than SAC-STM, but it ensures always better performance than ShrinkSTM and ATS-STM and for a maximum level of parallelism greater than 12 it is able to overcome TinySTM, too. By the graph we can clearly see that TinySTM, ShrinkSTM and ATS-STM exhibit a scalability problem with this workload. This phenomenon is avoided by SAC-STM (thanks to its proper thread activation/deactivation functionalities, which provide a means to control the negative effects associated with data contention) and by SwissTM.

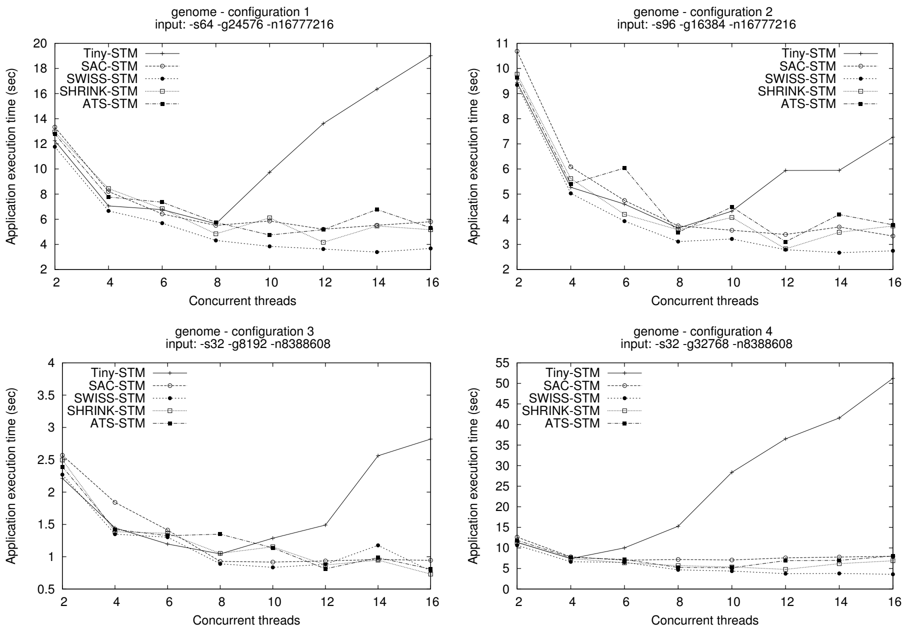


Figure 8.2: Application execution time - genome benchmark

The results of the tests with the genome benchmark are shown in Figure 8.2. For *configuration 1* and *configuration 2* SwissTM allows to obtain always the better performance, for all the concurrency level. For all the other approaches we can see that their performance are comparable up to 8 threads. With more threads, while the performance of TinySTM degrades, the other approaches, namely SAC-STM, ShrinkSTM and ATS-STM, ensure, independently of the number of available threads, an execution time comparable with the best one provided by TinySTM (i.e. 8 threads). With *configuration 3* the best execution time with TinySTM is achieved with 8 threads, after which the performance slightly decreases. About the other STM implementations we can see that SwissTM and SAC-STM ensure the same performance between 6 and 16 threads. Before 6 thread swissTM is able to ensure better performance than SAC-STM and comparable with the one obtainable with TinySTM. About ShrinkSTM and ATS-STM we can see that they provide the same performance offered by SwissTM except for the interval between 7 and 11 thread where they are not able to obtain the same performance ensured by both SAC-STM and SwissTM. For *configuration 4* TinySTM reaches its optimum using 4 threads, after the performance decreases. SwissTM ensures always the best performance. The remaining approaches are not able to ensure performance as good as the one reached by SwissTM, but they are very close.

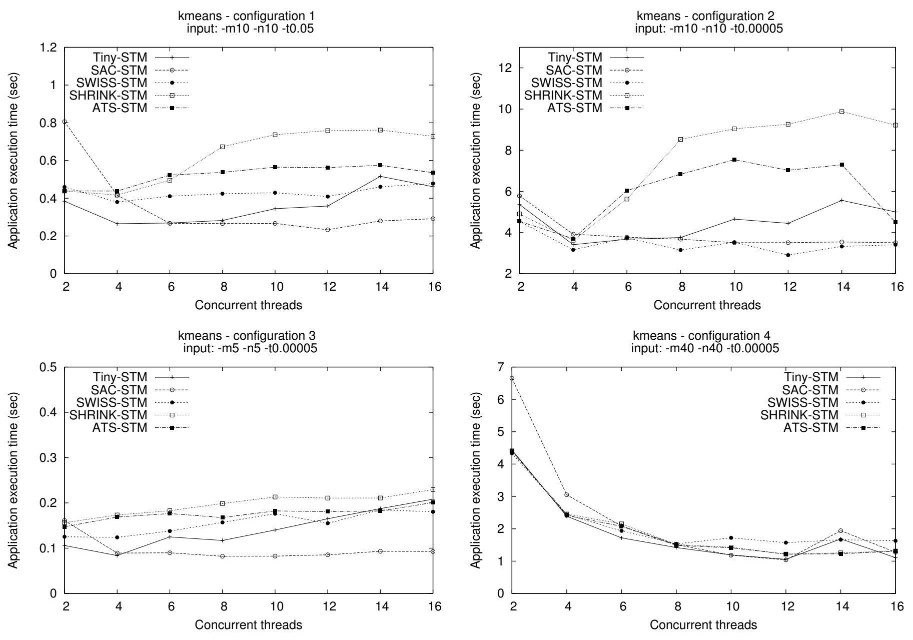


Figure 8.3: Application execution time - kmeans benchmark

In figure 8.3 we show the results obtained with the kmeans benchmark. For *configuration 1* we can see that ATS-STM and ShrinkSTM ensure the same execution time between 2 and 6 threads. For higher concurrency level the performance of ATS-STM remains stable until 16 threads. Conversely the performance of ShrinkSTM starts to get worse. About SwissTM, the graph shows that it ensures always better performance than ATS-STM and ShrinkSTM. The application execution time achieved with TinySTM decreases when increasing the number of used threads up to 4-6, while for greater values it increases. The performance offered by TinySTM are always better respect to the one offered by ATS-STM and ShrinkSTM. It reaches better performance than SwissTM until 12 thread. For greater values TinySTM and SwissTM offer very close performance. SAC-STM offers the best performance between 6 and 16 threads, but for concurrency levels lower than 4, it pays for the workload parameters sampling overhead, so its performance can't reach the ones offered by the other approaches. For *configuration 2*, ATS-STM, ShrinkSTM and TinySTM have a behaviour similar to the one already shown for configuration 1. About SAC-STM and SwissTM they provide the best performance between 6 and 16 threads (their performance are very similar). Between 1 and 5 threads SwissTM ensure the best performance, conversely SAC-STM pays again for the workload profile sampling overhead (less than in configuration 1). For *configuration 3* the results are again very similar to the ones obtained for configuration 1, with SAC-STM that is able to always reach the better performance for concurrency level between 4 and 16. About *configuration 4*, we can see that this scenario is totally different than the previous ones. For this configuration the number of available cores is not sufficient to reach the concurrency level that produces performance degradation. As we can see from the graph all the implementations allow to obtain similar performance. Only SwissTM for concurrency leves between 10 and 16 is not able to reach the same performance of the other STMs and SAC-STM for concurrency level between 1 and 6 pays again the costs for workload profile sampling.

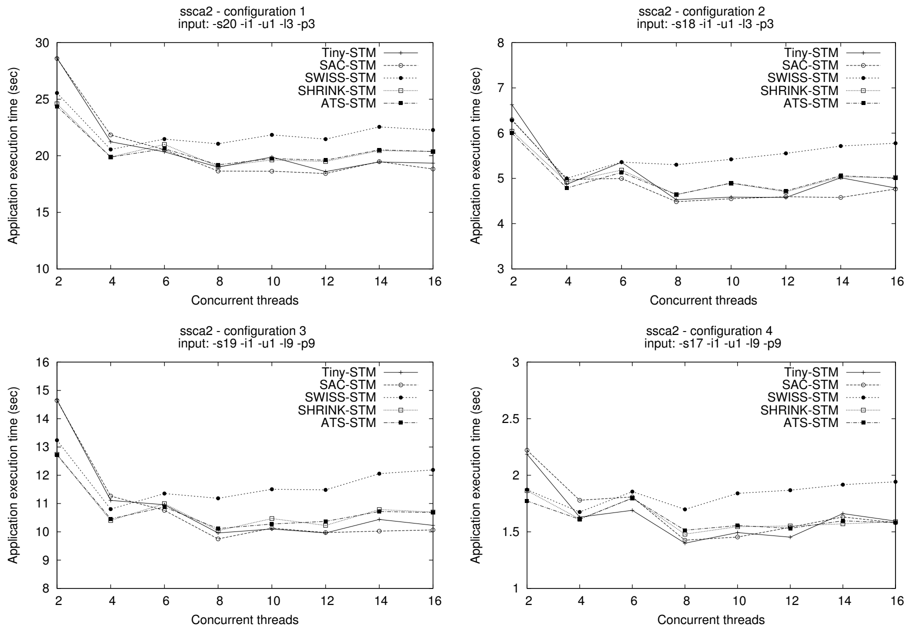


Figure 8.4: Application execution time - ssca2 benchmark

The Figure 8.4 shows test results for ssca2 benchmark. In all the tests we made with this benchmark we verified that the number of available cores is not sufficient to reach the concurrency level that produces performance degradation. As we can see from the graphs with this benchmark all the STM implementations, except SwissTM, ensure the same performance. SwissTM for concurrency level between 7 and 16 always provides performance between 5 % and 30 % worst than the other approaches.

The results of the tests with the vacation benchmark are shown in Figure

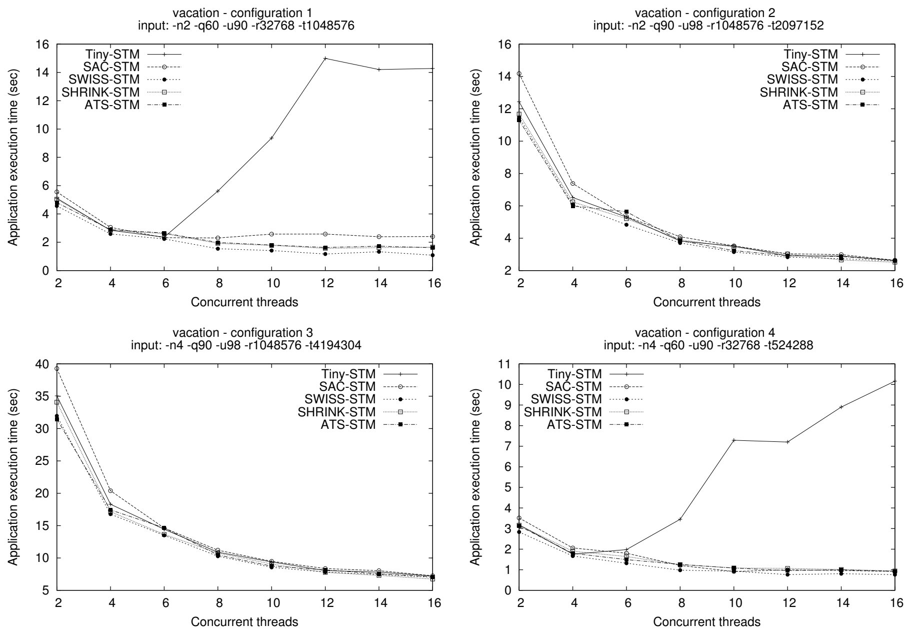


Figure 8.5: Application execution time - vacation benchmark

8.5. As we can see from the first and the fourth graphs, for *configuration 1* and *configuration 4* the application execution time achieved with TinySTM decreases when increasing the number of used threads respectively up to 6 and 4 while for greater values it drastically increases. All the other STM implementations have the same behaviour for all the level of parallelism ensuring the same application execution time. Only SAC-STM in the configuration 1 doesn't reach the same performance of the other approaches between 7 and 16 threads, but the difference is always less than 5%. For *configuration 2* and *configuration 3*, like for ssca2 benchmark, the number of available cores is not sufficient to reach the concurrency level that produces performance degradation. For this scenarios the graphs show that all the STM implementations allow to obtain the same performance.

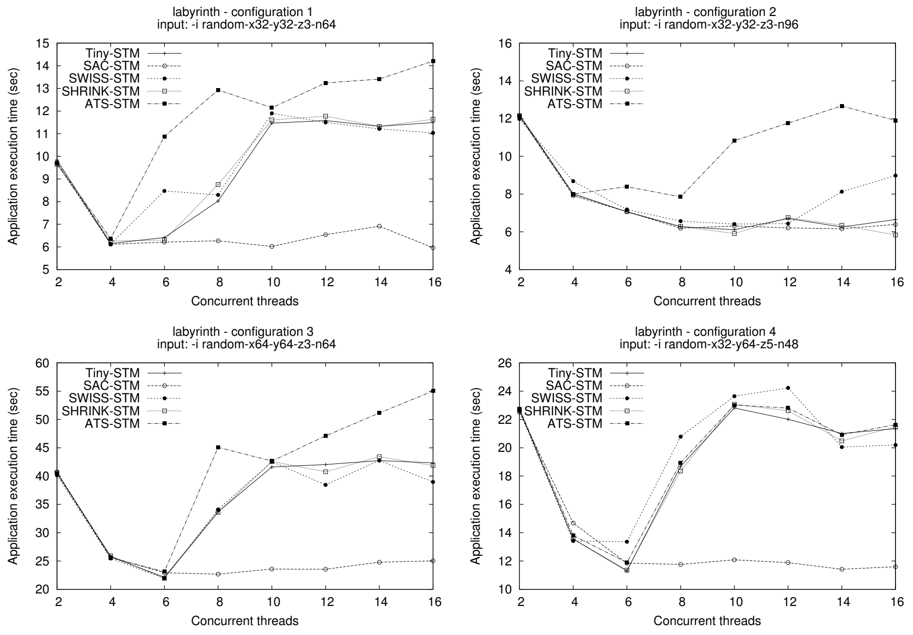


Figure 8.6: Application execution time - labyrinth benchmark

In figure 8.6 we show the results obtained with our modified version of the labyrinth benchmark. For the tests showed in figure 8.6 we iterate the application 1000 times ([1]). For *configuration 1* the application execution time achieved with all the STMs decreases when increasing the number of used threads up to 4. For greater values it starts to increase except for SAC-STM that is able to ensure always the optimal performance independently from the number of maximum available thread. About *configuration 2* we can see from the second graph that TinySTM, SAC-STM and ShrinkSTM have similar behaviour, the performance increases when increasing the number of used threads up to 10, than the performance remains stable. SwissTM has the same behaviour up to

[1]In our experimental evaluations we will use a modified version of the labyrinth benchmark. The original version executes a number of transactions not large enough large to allow to conclude some adaptivity steps. So we modified it including the benchmark inside a cycle that executes the application x times, where x is a configurable number.

12 thread, than its performance starts do decrease. About ATS-STM the graph shows that it is able to ensure the same performance of the other STMs up to 4 theads, for greater values the application execution time drastically increases. The *configuration 3* and *configuration 4* present results similar to configuration 1. The application execution time achieved with all the STMs decreases when increasing the number of used threads up to 6. For greater values it drastically increases except for SAC-STM that again is able to ensure always the optimal performance.

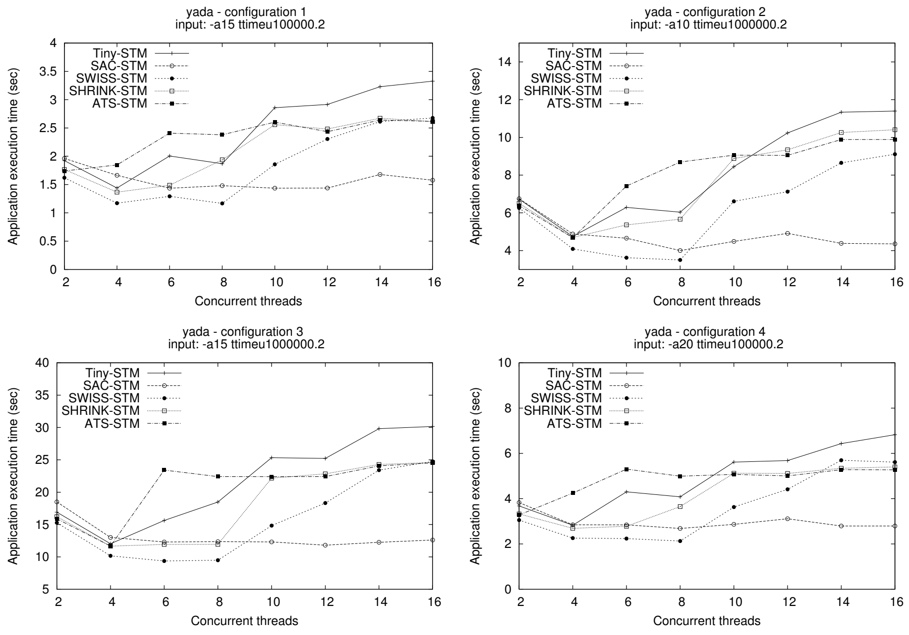


Figure 8.7: Application execution time - yada benchmark

Figure 8.7 shows the results obtained with the yada benchmark. For *configuration 1* and *configuration 4* the application execution time achieved with all the STMs, except ATS-STM, decreases when increasing the number of used threads up to 4. For greater values it starts to increase except for SAC-STM and

SwissTM. As we can see from the graph ATS-STM presents scalability problems, its application execution time in this configuration always increases, increasing the concurrency level. Between 4 and 8 thread only SAC-STM and SwissTM are able to avoid performance degradation, and SwissTM ensures better performance than SAC-STM. For values greater than 8 SwissTM is not still able to avoid performance degradation and as we can see from the graph its application execution time starts to increase. The only implementation that avoids performance degradation for any concurrency level is SAC-STM. For *configuration 2* the application execution time achieved with all the STMs decreases when increasing the number of used threads up to 4. For greater values it starts to drastically increase except for SAC-STM and SwissTM. For this two STMs the performance continues to increase up to 8 thread. For greater values the application execution time for SwissTM starts to increase. As we can see from the graph, the only implementation that allows to avoid degradation due to logical contention, for all the concurrency level, is again SAC-STM. The results for *configuration 3* are very similar to the ones of configuration 1 and configuration 4. The only differences are that ShrinkSTM is able to avoid performance degradation for concurrency level between 4 and 8 and ATS-STM presents lower scalability problems: its performance improves up to 4 threads but for greater values the application execution time starts again to increase.

The results of the tests with the bayes benchmark are shown in Figure 8.8. This application is not so stable (e.g. as already showed in [17]) but for completeness we decided to include it in our comparison trying to mitigate its instability executing an higher number of test and than reporting the mean values. For *configuration 1* the application execution with all the STM implementations decreases up to 6 threads, for greater values the application execution remains

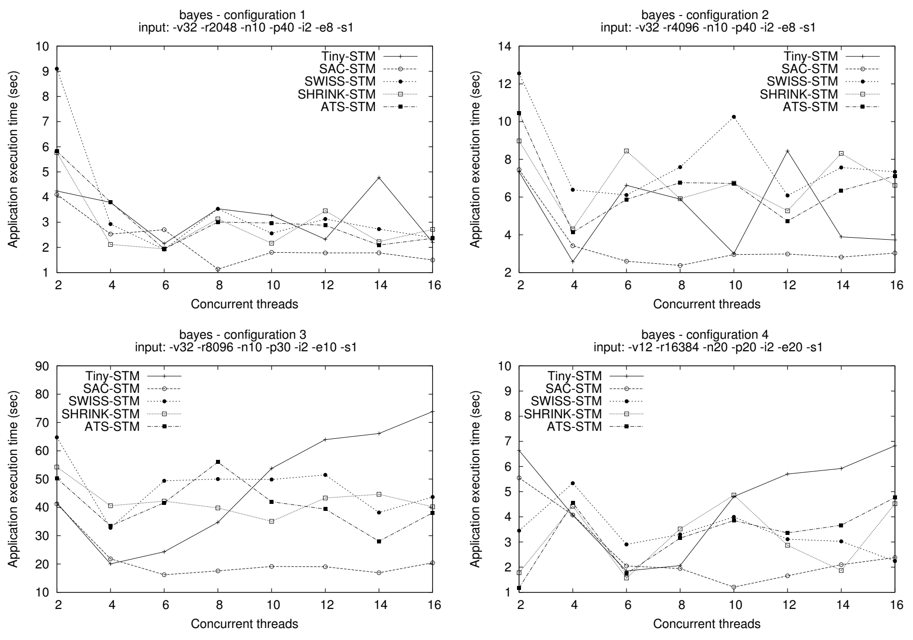


Figure 8.8: Application execution time - bayes benchmark

stable for all the STM except SAC-STM. For this implementation the application execution dime decreases until 8 and than it remains stable up to 16 threads on better values than the ones reachable with all the other implementations. For *configuration 2* and *configuration 4* bayes shows its instability. All the implementations, except SAC-STM, shows a big variability in the application execution time varying the concurrency level. SAC-STM, leveraging on its concurrency regulation mechanism, is able to ensure stable optimal performance independently of the maximum number of available threads. For *configuration 3* the application execution time decreases up to 4 threads for all the STMs but TinySTM and SAC-STM are able to ensure about 20% better performance than the other implementations. For higher concurrency level, the application execution time of TinySTM drastically increases. Instead SAC-STM is able to maintain optimal performance independently of the maximum number of available threads. About the other STM implementations, they presents high performance variability but their performance are always worst than SAC-STM.

By all the results, we can gather an important general result: all the solutions that don't dynamically adjust the concurrency level are prone to performance degradation when a not optimal number of threads is used to run the application. We have shown that solutions based on transaction scheduling mechanisms (like ATS-STM and ShrinkSTM) are not sufficient for capturing the intrinsic degree of parallelism and that adaptivity is an essential building block for creating STM systems which can offer optimal performance independently of the application workload. The results obtained with our adaptive solutions are very promising, as they shown that, in most of the cases, the performance achieved is, independently of the maximum number of concurrent threads used to execute the application, close to the best case when using a fixed (optimal) number of running threads. In particular, we observed that when an application is executed with an overestimated number of concurrent threads, our self-adjusting STMs prove to be able to reduce the concurrency level so to avoid the typical performance degradation experienced with traditional (non-self adjusting) STM systems. Moreover, being orthogonal to the other approaches, they can be used in combination with all the existing implementations allowing to further optimize their performance.

Chapter 9

Conclusions

Software Transactional Memory is a programming paradigm for parallel/concurrent applications that represents an easy-to-use alternative to traditional lock-based synchronization mechanisms. By leveraging on the concept of atomic transactions, historically used in the field of database systems, STMs relieve programmers from the burden of explicitly writing complex, error-prone thread synchronization code. STMs provide a simple and intuitive programming model, where programmers wrap critical-section code within transactions, thus removing the need for using fine-grained lock-based synchronization approaches. Programmers' productivity is therefore improved, while not sacrificing the advantages provided by high parallelism, thus avoiding any loss in performance typically associated with serial execution scenarios, or with cases where an easy to program, coarse-grained locking approach is used.

Thanks to the diffusion of the multi-core systems (today even entry-level desktop and laptop machines are equipped with multiple processors and/or CPU-cores) the relevance of the STM paradigm has significantly grown. Together with the simplification of the development process, an aspect that is central for the success, and the further diffusion of the STM paradigm relates to the actual level of performance it can deliver. As for this aspect, one core issue to cope with in STM is related to exploiting parallelism while also avoiding thrashing phenomena due to excessive transaction rollbacks, caused by excessive contention on logical resources, namely concurrently accessed data portions.

In this thesis we dealt with the problem of regulating the concurrency level in STM based applications with the aim to optimize the performance, avoiding penalties due to the wrong choice of the concurrency level selected for running the applications.

We presented a suite of techniques for dynamic concurrency regulation based on three different approaches:

- a Machine Learning based approach;
- an analytical modelling based approach;
- a grey box approach that mix the previous ones (allowing to chase the best of the two methodologies by tackling the drawbacks intrinsic in each of them).

The first approach uses Neural Network (NN) to approximate the function of the mean transactional wasted time, that is the time spent by a thread to execute transactional code that will be aborted. The predictions of the NN are then used to evaluate the mean application throughput varying the concurrency level. In this way it is possible to determine the number of threads that allows to maximize the performance.

The second approach uses a parametric analytical model to approximate the transaction abort probability function. This approximated function is then used to evaluate again the mean transactional wasted time. The instantiation of the parameters of the model can be actuated via a light regression process based on a few samples related to the run-time behavior of the application. Also, the model does not rely on any strong assumption in relation to the application profile, hence being usable in generic application contexts.

The third approach combines the previous ones taking the best from each of them. The results of the combination is a methodology for fast construction of a highly reliable performance model for the determination of the optimal level of concurrency for STM-based applications. This is relevant in generic contexts also including the Cloud, where the need for deploying new applications (or applications with reshuffling in their execution profile), while also promptly determining the system configurations allowing optimized resource usage, is very common.

All the developed approaches are able to cope with cases where the actual application workload profile can change over time, such as when the (average) size of the data-set accessed by the transactional code in read or write mode changes over time (e.g. according to a phase-behavior).

Moreover we presented an innovative approach for dynamically selecting the input features to be exploited by the Machine Learning based performance model. The approach relies on runtime analysis of variance and correlation of workload characterization parameters, and on feedback control on the quality of performance prediction achieved with shrunk sets of features. From the reported experiments we showed that this approach is able to reduce the overhead for features sampling, minimizing its impact on application performance.

Together with the performance prediction models we also implemented a suite of prototypes of concurrency regulation architectures, integrated with the TinySTM open source framework, which exploit the developed solutions to dynamically tune the number of threads used for running the application and to minimize the workload parameters sampling overhead.

Using these prototypes we did a detailed experimental evaluation executing all the applications of the STAMP benchmark suite and comparing our solutions with other adaptive (ATS-STM and ShrinkSTM) and standard STM implementations (TinySTM and SwissTM). The obtained results show the effectiveness of all the proposed approaches, more in detail the performance achieved is, independent of the maximum number of concurrent threads of the application, close to the best case when using a fixed (optimal) number of running threads with standard version of TinySTM. In particular, we observed that when an application is executed with an underestimated number of concurrent threads our-self adjusting STMs are able to ensure performance very close to that offered by standard STMs (proving the low overhead paid by our concurrency regulation mechanisms) and when an overestimated number of concurrent threads is used, our solutions are able to properly regulate the concurrency level so to avoid the performance degradation experienced with traditional STM systems.

Encouraged by the effectiveness of the developed solutions and exploiting the diffusion of hardware transactional memory (HTM) architectures (e.g. Intel processors with TSX extension) we started to evaluate the applicability of our adaptive approaches to HTM based applications. During our first study we verified that using HTM the primary transaction abort reason changes. The contention on shared data is no longer the main abort cause: for HTM based applications aborts are mainly due to the limited size of the transactional cache and to other architectural reasons. Taking this aspects into account, we started to evaluate the effectiveness of our approaches and we started to deploy new solutions explicitly tailored for HTM based applications.

Finally we successfully applied our Machine Learning performance prediction methodology in the context of transactional Data Grid platforms deployed in Cloud Computing environments. We developed a suite of self-regulating architectures that are able to automatically reconfigure the Data Grid Platform (in terms of both the well suited amount of cache servers, and the well suited degree of replication of the data-objects) with the aim of guarantee specific throughput or latency values (such as those established by some SLA), under some specific workload profile/intesity, while minimizing at the same time the cost for the cloud infrastructure [78, 79].

Appendix A

Neural Networks

A Neural Network (NN) is a machine learning method [38] having the ability to approximate various kinds of functions, including real-valued and discrete-valued ones. Inspired to the neural structure of the human brain, a NN consist of a set of interconnected processing elements, commonly referred to as neurons, which cooperate to compute a specific function, so that, provided a given input, the NN can be used to estimate the output the function. Each element of the NN, in turns, calculates a (simpler) function, called transfer function. Different types of processing elements have been designed, each one calculating a specific transfer function. Commonly used elements are:

- *perceptron*: an element that takes as input a vector of real valued inputs, calculates a linear combination of them and then outputs 1 if the result is greater that a given threshold, -1 otherwise. The output of the perceptron is:

$$o(\vec{x}) = sign(\vec{w} \cdot \vec{x}) \tag{A.1}$$

  where $\vec{x}$ is the perceptron input vector and $\vec{w}$ is the weight vector.

- *linear unit*: an element that takes as input a vector of real values and outputs the weighted sum of inputs plus a bias term. The output of the linear input can be written as:

$$o(\vec{x}) = (\vec{w} \cdot \vec{x}) \tag{A.2}$$

- *sigmoid unit*: it is similar to perceptron, but it has a smoothed, differentiable threshold function. So the output of this element is a not linear, differentiable function of its inputs. The sigmoid unit computes output as:

$$o(\vec{x}) = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x})}} \tag{A.3}$$

Different type of neurons can be connected using weighted arches to build an arbitrary complex NN. Neurons constituting the network are grouped into layers. The minimum number of layers necessary to build a NN is three: (i) an input layer, (ii) a hidden layer and (iii) an output layer. An example of a three layer fully connected ([1]) NN is given in Figure A.1. More complex network can be builded increasing the number of hidden layer. The type and the number of neurons and the number of hidden layers are parameters that depends form the specific application.

In order to approximate a function f, a so-called learning algorithm can be used. Basically, a learning algorithm computes the weight associated to arches that connect the network units. By relying on a learning algorithm, a NN can be trained exploiting a set $\{(\mathbf{i}, \mathbf{o})\}$ (training set) of samples, where, for each

[1]A fully conected neural network is an artificial neural network where each node of each layer is connected with each node of the adiacent layers

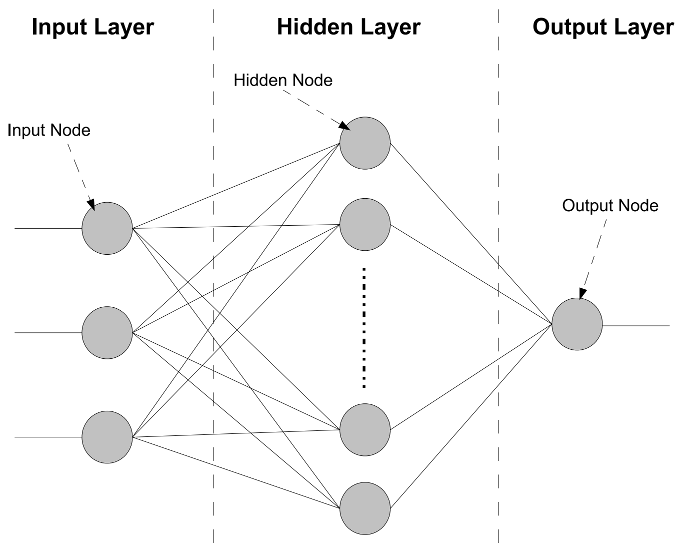


Figure A.1: Example of a three layer fully connected neural network

sample $(\mathbf{i}, \mathbf{o})$, it is assumed that $\mathbf{o} = f(\mathbf{i}) + \delta$, where δ is a random variable (also said *noise*).

Essentially, each sample provides the training algorithm information about the existent relation between the input and the output of the function f. Usually, a learning algorithm works according to an iterative procedure, where in each iteration step it performs the following computation:

- for each sample $(\mathbf{i}, \mathbf{o})$ of the training set, it calculates the output associated to the input $\mathbf{i}$ using the NN;
- it compares the obtained output with the associated output of the sample, i.e., $\mathbf{o}$, determining the error;
- the algorithm modifies the weights of the NN arches with the aim to minimize the overall error for the whole training set.

The iterative procedure can be stopped after a given number of steps have been executed or when the error is below a given threshold.

Various learning algorithms have been proposed. The design of these algorithms depends on the NN topology and on the specific type of computation the NN is intended for. In order to approximate arbitrary complex real valued functions, a multilayer not acyclic sigmoid-based neural network can be used ([80, 81, 82]). To train this type of NN with a fixed set of elements and interconnections, a commonly used learning algorithms is the the Backpropagation algorithm [69]. Basically, this algorithm calculates the weights of the NN exploiting gradient descent to attempt to minimize the mean squared error between the NN output values and the output values of the training set.

Appendix B

The STAMP Benchmark Suite

In this chapter we briefly describe the applications included inside the STAMP benchmark suite that we will use for the experimental evaluation of the solutions proposed inside this thesis. A detailed description of all the applications can be found in [20]. We chose to make our experimental evaluation using STAMP because it is the standard benchmark suite used for STM testing and it provides a wide and complete coverage of various scenarios providing a very good approximation of real application workloads:

- **intruder**: it is an application which implements a signature-based network intrusion detection systems (NIDS) that scans network packets for matches against a known set of intrusion signatures. In particular, it emulates Design 5 of the NIDS described in [83]. Three analysis phases are carried on in parallel: *capture*, *reassembly*, and *detection*. The capture and reassembly phases are each enclosed by transactions, which are relatively short and show a contention level which is either moderate or high, depending on how often the reassembly phase rebalances its tree. Overall, the total amount of time spent in the execution of transactions is relatively moderate. It is a good candidate application to measure the performance when transactional memory middlewares must manage a high level of logical contention.

- **kmeans**: it is a transactional implementation of a partition-based clustering algorithm [84]. A cluster is represented by the mean value of all the objects it contains, and during the execution of this benchmark the mean points are updated by assigning each object to its nearest cluster center, based on Euclid distance. This benchmark relies on threads working on separate subsets of the data and uses transactions in order to assign portions of the workload and to store final results concerning the new centroid updates. Given the reduced amount of shared data structures being updated by transactions, in this benchmark it is more likely to incur in logical contention when a larger number of threads is used for the computation. Therefore, this application benchmark is a good candidate to study how changing workload dynamics can affect performance when scaling up a transactional application.

- **yada**: it implements Ruppert's algorithm for Delaunay mesh refinement [85], which is a key step used for rendering graphics or to solve partial differential equations using the finite-element method. This benchmark discretizes a given domain of interest using triangles or thetraedra, by iteratively refining a coarse initial mesh. In particular, elements not satisfying quality constraints are identified, and replaced with new ones, which in turn might not satisfy the constraints as well, so that a new replacement phase must be undertaken. This benchmark shows a high level of parallelism, due to the fact that elements which are distant in the mesh do not interfere with each other, and operations enclosed by transactions involve only updates of the shared mesh representation and cavity expansion. The overall execution time of this benchmark is relatively long, showing a high duration of transaction operations and a significantly higher number of memory operations. Overall, this is a good candidate application to measure the performance when transactional memory middlewares must manage a medium level of logical contention.

- **vacation**: it is an application that simulates a travel reservation system powered by a centralized database. The workload is generated by several client threads that submit requests to the database via the system's transaction manager. The database contains only four tables: one for cars, one for rooms, one for flights and one for customers. The first three have columns representing a unique ID number, the already reserved quantity, the total available quantity, and the price. The forth table tracks the reservations made by each customer and the total price of the reservations they made. The tables are implemented using Red-Black trees data structures. This application presents a logical contention level that can vary from low to medium depending on the input parameters provided at application start-up.

- **labyrinth**: it is an application that finds the shortest-distance paths between pairs of starting and ending points inside a maze. It uses the routing Lee's algorithm [86]. The maze is represented as a grid where each grid point can contain connections to adjacent grid points (not in diagonal). The algorithm searches for a shortest path between the start and end points of a connection by performing a breadth-first search and labelling each grid point with its distance from the start. This searching phase will eventually reach the end point if a free path inside the maze exists.

Then a second trace-back phase forms the connection by following any path with a decreasing distance. This algorithm guarantees to find the shortest path between the start and the end point but, when multiple paths are made, one path may block another. The implementation of a transactional version of this algorithm was made using the techniques described in [87]. Overall, this is a good candidate application to measure the performance when transactional memory middlewares must manage a high level of logical contention.

- **ssca2**: it is an application that implements one of the four kernel defined in [88]. In this work a set of four kernels that operate on a large, directed, weighted multi-graph are defined. This kernels are used in application ranging from security to computational biology. More in detail, the ssca2 application provided in STAMP implements the Kernel 1 that constructs an efficient graph data structure using adjacency and auxiliary arrays. In the transactional version there are threads that concurrently add nodes to the graph and transactions are used to protect access to the adjacency arrays. This operation is relatively small so not much time is spent in transactions. So this application is characterized by short transactions and small sized transactional read set and write set. The large number of graph nodes lead to infrequent concurrent updates of the same adjacency list so the amount of contention is also relatively low.

- **genome**: it is an application that implements a gene sequencing program. Genome assembly is a process that take a large number of DNA segments and match them to reconstruct the original source genome. The algorithm for gene sequences implemented in this application is divided in three phases. Since there is a relatively large number of DNA segments, there are often many duplicates, so the first phase removes duplicate segments by using hash-set creating a set of unique segments. In the second phase each thread tries to remove a segment from a global pool of unmatched segments and add it to its partition of currently matched segments. During segments matching the Rabin-Karp string search algorithm [89] is used to seed up the comparison (the cycles are prevented by tracking starts and ends of matched chains). In the third phase the algorithm builds the sequence. Transactions are used in the first two phases of the benchmark. Transactions are of moderate length and have moderate read and write set sizes. There is low contention.

- **bayes**: it is an application that implements an algorithm for the learning of Bayesian networks, that are used to represent probability distribution for a set of variables. A Bayesian network is a direct acyclic graph, where each node is used to represent a variable and each edge is used to represent a conditional dependence. A Bayesian network is able to represent all of the probability distribution recording the conditional independences among variables (the lack of edges between nodes implies conditional independence between the variable represented by the nodes). Typically, the probability distributions and the conditional dependences among them are known or solvable for a human, thus Bayesian networks are learned from data. The algorithm implemented inside the application is based on a hill-climbing strategy that uses both local and global search, similar to the technique described in [90]. For efficient probability distribution estimations, the adtree data structure described in [91] is used. This benchmark presents low logical contention.

Application	**Tx length**	**RS/WS size**	**Tx time**	**Contention**
bayes	Long	Large	High	High
labyrinth	Long	Large	High	High
intruder	Short	Medium	Medium	High
kmeans	Short	Small	Low	Low
yada	Long	Large	High	Medium
vacation	Medium	Medium	High	Low/Medium
ssca2	Short	Small	Low	Low
genome	Medium	Medium	High	Low

Table B.1: Qualitative summary of STAMP application's runtime transactional characteristics

In table B.1 we report a qualitative summary of each application's runtime transactional characteristics (length of transactions, read set and write set size, time spent executing transactional code, amount of logical contention). A most detailed quantitative characterization can be found in [20]. As we can see from the table, STAMP offers a lot of combinations that allow to cover a wide range of transactional execution behaviors. In this way it allows to stress all the aspects of the evaluated TM systems and so it represents a good test bed for evaluating the performance of STM platforms.

Bibliography

[1] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM. (cited on pages 7, 13).

[2] WMware. vFabric GemFire XX. http://www.vmware.com/products/vfabric-gemfire/overview.html. (cited on page 7).

[3] Oracle. Orache Coherence. http://www.oracle.com/technetwork/middleware/coherence/overview/index.html, 2011. (cited on page 7).

[4] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Operating System Review*, 44(2):35–40, April 2010. (cited on page 7).

[5] Paolo Romano, Luis Rodrigues, Nuno Carvalho, and Joäo Cachopo. Cloud-tm: Harnessing the cloud with distributed transactional memories. *SIGOPS Operating System Review*, 44(2):1–6, April 2010. (cited on page 8).

[6] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, DISC 2006, pages 194–208, 2006. (cited on pages 8, 15, 24, 32, 137).

[7] Pascal Felber, Christof Fetzer, and Torvald 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, pages 237–246. ACM, 2008. (cited on pages 8, 32, 170).

[8] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Computer Architecture News*, 21(2):289–300, may 1993. (cited on pages 8, 13, 32).

[9] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 141–150, New York, NY, USA, 2009. ACM. (cited on pages 8, 32).

[10] Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a scalable software transactional memory. In *4th ACM SIGPLAN Workshop on Transactional Computing*, TRANSACT09, feb 2009. (cited on pages 8, 32).

[11] Qingping Wang, Sameer Kulkarni, John V. Cavazos, and Michael Spear. Towards applying machine learning to adaptive transactional memory. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, 2011. (cited on pages 8, 32).

[12] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 169–178, New York, NY, USA, 2008. ACM. (cited on pages 8, 32, 167).

[13] Aleksandar Dragojević, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC, pages 7–16. ACM, 2009. (cited on pages 8, 32, 33, 167).

[14] Pierangelo Di Sanzo, Bruno Ciciani, Roberto Palmieri, Francesco Quaglia, and Paolo Romano. On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking. *Performance Evaluation*, 69(5):187 – 205, 2012. (cited on pages 9, 10, 38, 42, 47, 112, 136, 150).

[15] Diego Didona, Pascal Felber, Derin Harmanci, Paolo Romano, and Jrg Schenker. Identifying the optimal level of parallelism in transactional memory applications. *Networked Systems*, 7853:233–247, 2013. (cited on pages 9, 39, 40, 42, 150).

[16] Zhengyu He and Bo Hong. Modeling the run-time behavior of transactional memory. In *Proceeding of the 18th IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*, MASCOTS, pages 307 –315, aug. 2010. (cited on pages 9, 10, 35, 41, 136, 150).

[17] Aleksandar Dragojević and Rachid Guerraoui. Predicting the scalability of an stm: A pragmatic approach. In *5th ACM SIGPLAN Workshop on Transactional Computing*, April 2010. (cited on pages 9, 35, 36, 41, 182).

[18] Maria Couceiro, Pedro Ruivo, Paolo Romano, and Luis Rodrigues. Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation. *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 0:1–12, 2013. (cited on page 10).

[19] Pascal Felber, Christof Fetzer, and Torvald 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*, pages 237–246, New York, NY, USA, 2008. ACM. (cited on pages 11, 15, 22, 111).

[20] Chi C. Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46, Seattle, WA, USA, 2008. (cited on pages 11, 109, 112, 123, 195, 200).

[21] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking concurrency. *Queue*, 4(10):24–33, December 2006. (cited on page 14).

[22] Pascal Felber, Christof Fetzer, Rachid Guerraoui, and Tim Harris. Transactions are back—but are they the same? *SIGACT News*, 39(1):48–58, March 2008. (cited on page 14).

[23] Roberto Palmieri, Francesco Quaglia, Paolo Romano, and Nuno Carvalho. Evaluating database-oriented replication schemes in software transactional memory systems. In *IPDPS Workshops*, pages 1–8. IEEE, 2010. (cited on page 14).

[24] P. Romano, N. Carvalho, and L. Rodrigues. Towards distributed software transactional memory systems. Watson Research Labs, Yorktown Heights (NY), USA, Sept. 2008. (cited on page 14).

[25] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM. (cited on pages 15, 168).

[26] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, December 1992. (cited on page 15).

[27] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Notices*, 41(10):253–262, October 2006. (cited on page 15).

[28] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 253–262, New York, NY, USA, 2006. ACM. (cited on page 15).

[29] Microsoft. http://msdn.microsoft.com/en-us/devlabs. (cited on page 15).

[30] Intel. http://software.intel.com/en-us/blogs/2013/06/07/resources-about-intel-transactional-synchronization-extensions. (cited on page 15).

[31] Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the 20th International Symposium on Distributed Computing*, DISC, pages 179–193. Springer, 2006. (cited on page 18).

[32] Mohammad Ansari, Christos Kotselidis, Mikel Lujan, Chris Kirkham, and Ian Watson. On the performance of contention managers for complex transactional memory benchmarks. In *Proceedings of the 2009 Eighth International Symposium on Parallel and Distributed Computing*, ISPDC '09, pages 83–90, Washington, DC, USA, 2009. IEEE Computer Society. (cited on pages 19, 21).

[33] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010. (cited on page 19).

[34] Márcio Bastos Castro, Kiril Georgiev, Vania Marangozova-Martin, Jean-François Méhaut, Luiz Gustavo Fernandes, and Miguel Santana. Analysis and tracing of applications based on software transactional memory on multicore architectures. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP, pages 199–206, Los Alamitos, CA, USA, 2011. IEEE Computer Society. (cited on page 24).

[35] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. *SIGPLAN Notices*, 44:155–165, June 2009. (cited on pages 24, 35, 167).

[36] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why stm can be more than a research toy. *Communications of the ACM*, 54(4):70–77, April 2011. (cited on page 24).

[37] Paolo Romano and Matteo Leonetti. Self-tuning batching in total order broadcast protocols via analytical modelling and reinforcement learning. In *Proceedings of the IEEE International Conference on Computing, Networking and Communications*, ICNC, Washington, DC, USA, 2012. IEEE Computer Society. (cited on page 29).

[38] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1 edition, 1997. (cited on pages 29, 57, 65, 191).

[39] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, March 1986. (cited on page 29).

[40] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern classification*. Pattern Classification and Scene Analysis. Wiley, 2001. (cited on page 29).

[41] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, September 1995. (cited on page 29).

[42] G.E. Hinton and T.J. Sejnowski. *Unsupervised Learning: Foundations of Neural Computation*. A Bradford Book. MIT Press, 1999. (cited on page 29).

[43] Peter Auer and Ronald Ortner. Ucb revisited: Improved regret bounds for the stochastic multi-armed bandit problem. *Periodica Mathematica Hungarica*, 61(1):5565, 2010. (cited on page 29).

[44] Daniel M. Berry. *Bandit problems: Sequential Allocation of Experiments*. Springer, 1985. (cited on page 29).

[45] R.S. Sutton and A.G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning. MIT Press, 1998. (cited on page 29).

[46] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47:235–256, May 2002. (cited on page 29).

[47] T. Ishikida and P. Varaiya. Multi-armed bandit problem revisited. *Journal of Optimization Theory and Applications*, 83:113–154, October 1994. (cited on page 29).

[48] Niels Reimer, Stefan U. Hngen, and Walter F. Tichy. Dynamically adapting the degree of parallelism with reflexive programs. In *IRREGULAR*, volume 1117 of *Lecture Notes in Computer Science*, pages 313–318. Springer, 1996. (cited on page 30).

[49] Hans-Ulrich Heiss and Roger Wagner. Adaptive load control in transaction processing systems. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB, pages 47–54, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc. (cited on page 30).

[50] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, Erich M. Nahum, and Adam Wierman. How to determine a good multi-programming level for external scheduling. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE, page 60. IEEE Computer Society, 2006. (cited on page 31).

[51] Mohammed Abouzour, Kenneth Salem, and Peter Bumbulis. Automatic tuning of the multiprogramming level in Sybase SQL Anywhere. In *Workshops co-located with International Conference on Data Engineering*, pages 99–104. IEEE, 2010. (cited on page 31).

[52] Qingping Wang, Sameer Kulkarni, John Cavazos, and Michael Spear. A transactional memory with automatic performance tuning. *ACM Transactions on Architecture and Code Optimization*, 8(4):54:1–54:23, January 2012. (cited on page 32).

[53] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *Proceedings 4th International Conference on High Performance and Embedded Architectures and Compilers*, HIPEAC, pages 4–18, jan 2009. Springer-Verlag Lecture Notes in Computer Science volume 5409. (cited on page 34).

[54] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982. (cited on page 34).

[55] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd annual Symposium on Principles of Distributed Computing*, PODC, pages 92–101, New York, NY, USA, 2003. ACM. (cited on page 35).

[56] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the 19th annual ACM Symposium on Parallel algorithms and Architectures*, SPAA, pages 221–228, New York, NY, USA, 2007. ACM. (cited on page 35).

[57] N. I. Naum Ilich Akhiezer. *Theory of approximation*. Dover Publications, New York, 1992. Translation of: Lektsii po teorii approksimatsii. (cited on page 37).

[58] Donald E. Porter and Emmett Witchel. Understanding transactional memory performance. In *Proceedings of the IEEE International Symposium on Performance Analysis of Software Systems*, pages 97–108, March 2010. (cited on pages 38, 39, 42).

[59] Armin Heindl and Gilles Pokam. An analytic framework for performance modeling of software transactional memory. volume 53, pages 1202–1214, New York, NY, USA, June 2009. Elsevier North-Holland, Inc. (cited on pages 38, 39, 42).

[60] Armin Heindl, Gilles Pokam, and Ali-Reza Adl-Tabatabai. An analytic model of optimistic software transactional memory. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 153–162. IEEE, 2009. (cited on pages 38, 39, 42).

[61] Diego Didona, Pascal Felber, Derin Harmanci, and Paolo Romano. Elastic scaling for transactional memory: From centralized to distributed architectures (poster). In *4th Usenix Workshop on Hot Topics in Parallelism*, HOTPAR, June 2012. (cited on pages 39, 40, 42).

[62] Diego Didona, Paolo Romano, Sebastiano Peluso, and Francesco Quaglia. Transactional auto scaler: Elastic scaling of in-memory transactional data grids. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC, pages 125–134, New York, NY, USA, 2012. ACM. (cited on pages 39, 40, 42, 141).

[63] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *Proceedings of the 14th International Conference on Parallel Processing*, Euro-Par, pages 719–728, Berlin, Heidelberg, 2008. Springer-Verlag. (cited on page 40).

[64] Marcio Castro, Luis Fabricio Wanderley Goes, Christiane Pousa Ribeiro, Murray Cole, Marcelo Cintra, and Jean-Francois Mehaut. A machine learning-based approach for thread mapping on transactional memory applications. In *Proceedings of the 18th International Conference on High Performance Computing*, HIPC, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society. (cited on page 40).

[65] Márcio Bastos Castro, Luís Fabrício Wanderley Góes, Luiz Gustavo Fernandes, and Jean-François Méhaut. Dynamic thread mapping based on machine learning for transactional memory applications. In *Proceedings of the 18th International Conference on Parallel Computing*, Euro-Par, pages 465–476, Berlin, Heidelberg, 2012. Springer-Verlag. (cited on page 40).

[66] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006. (cited on pages 55, 100).

[67] http://leenissen.dk/fann/wp. (cited on page 65).

[68] Stuart J. Russell, Peter Norvig, John F. Candy, Jitendra M. Malik, and Douglas D. Edwards. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. (cited on pages 69, 125).

[69] A.E. Bryson and Y.C. Ho. *Applied Optimal Control: Optimization, Estimation, and Control*. Halsted Press book. Taylor & Francis, 1975. (cited on pages 69, 194).

[70] David E. Rumelhart and Ronald J. Williams Geoffrey E. Hinton. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. (cited on page 69).

[71] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: Scheduling-based Collision Avoidance and Resolution for Software Transactional Memory. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing*, PODC, pages 125–134, New York, NY, USA, 2008. ACM. (cited on pages 87, 150).

[72] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985. (cited on page 89).

[73] Philip S. Yu, Daniel M. Dias, and Stephen S. Lavenberg. On the analytical modeling of database concurrency control. *Journal of the ACM*, 40(4):831–872, September 1993. (cited on pages 110, 115).

[74] In Kyung Ryu and Alexander Thomasian. Performance analysis of centralized databases with optimistic concurrency control. *Performance Evaluation*, 7(3):195–211, 1987. (cited on page 110).

[75] Neil J. Gunther. *Guerrilla capacity planning - a tactical approach to planning for highly scalable applications and services*. Springer, 2007. (cited on page 129).

[76] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979. (cited on page 168).

[77] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, PODC, pages 338–339, New York, NY, USA, 2007. ACM. (cited on page 169).

[78] Pierangelo Di Sanzo, Diego Rughetti, Bruno Ciciani, and Francesco Quaglia. Providing transaction class-based QoS in in-memory data grids via machine learning. In *Proceedings of the 3nd IEEE Symposium on Network Cloud Computing and Applications*, NCCA. IEEE Computer Society, feb 2014. (cited on page 189).

[79] Pierangelo Di Sanzo, Diego Rughetti, Bruno Ciciani, and Francesco Quaglia. Auto-tuning of cloud-based in-memory transactional data grids via machine learning. In *Proceedings of the 2nd IEEE Symposium on Network Cloud Computing and Applications*, NCCA. IEEE Computer Society, dec 2012. (cited on page 189).

[80] G. Cybenko. Approximation by superpositions of a sigmoidal function. volume 2, pages 303–314, 1989. (cited on page 194).

[81] K. Funahashi. On the approximate realization of continuous mappings by neural networks. volume 2, pages 183–192, Oxford, UK, UK, May 1989. Elsevier Science Ltd. (cited on page 194).

[82] D.R. Hush and B.G. Horne. Progress in supervised neural networks. volume 10, pages 8 – 39, 1993. (cited on page 194).

[83] Bart Haagdorens, Tim Vermeiren, and Marnix Goossens. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *Proceedings of the 5th International Conference on Information Security Applications*, WISA, pages 188–203. Springer-Verlag, 2005. (cited on page 195).

[84] James C. Bezdek. *Pattern Recognition with Fuzzy Objective Function Algorithms.* Kluwer Academic Publishers, Norwell, MA, USA, 1981. (cited on page 196).

[85] J. Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18(3):548–585, 1995. (cited on page 196).

[86] C. Lee. An algorithm for path connections and its applications. *IRE Transactions On Electronic Computers*, 1961. (cited on page 197).

[87] Ian Watson, Chris C. Kirkham, and Mikel Luján. A study of a transactional parallel routing algorithm. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 388–398. IEEE Computer Society, 2007. (cited on page 198).

[88] David A. Bader and Kamesh Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing*, HiPC'05, pages 465–476, Berlin, Heidelberg, 2005. Springer-Verlag. (cited on page 198).

[89] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987. (cited on page 199).

[90] David Maxwell Chickering, David Heckerman, and Christopher Meek. A bayesian approach to learning bayesian networks with local structure. *Computer Research Repository*, abs/1302.1528, 2013. (cited on page 199).

[91] Andrew Moore and Mary Soon Lee. Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research*, 8:67–91, 1997. (cited on page 199).