KTH ROYAL INSTITUTE
OF TECHNOLOGY

Doctoral Thesis in Information and Communication Technology

Augmenting Transactional Memory with the Future Abstraction

JINGNA ZENG

Stockholm, Sweden 2020

Augmenting Transactional Memory with the Future Abstraction

JINGNA ZENG

Academic Dissertation which, with due permission of the KTH Royal Institute of Technology, is submitted for public defence for the Degree of Doctor of Philosophy on Thursday the 22nd of October 2020, from 9:00 to 12:00 at Ka-Sal C (Sven-Olof Öhrvik), Kistagången 16, Electrum 1, floor 2, KTH Kista.

Doctoral Thesis in Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden 2020

TRITA-EECS-AVL-2020:50
ISBN 978-91-7873-654-6

Printed by: Universitetsservice US-AB, Sweden 2020

Co-funded by the
Erasmus+ Programme
of the European Union

Augmenting Transactional Memory with the Future Abstraction

JINGNA ZENG

Doctoral thesis in Information and Communication Technology
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden 2020
and
Doctoral thesis in Information Systems and Computer Engineering
Universidade de Lisboa, Instituto Superior Técnico
Lisbon, Portugal 2020

TRITA-EECS-AVL-2020:50
ISBN 978-91-7873-654-6

KTH School of Electrical
Engineering and Computer Science
SE-164 40 KISTA
Stockholm
Sweden

Akademisk avhandling som med tillstånd av Kungliga Tekniska Högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i Informations-och kommunikationsteknik på torsdagen den 22 oktober 2020 klockan 9:00 i Sal C, Electrum, Kungliga Tekniska Högskolan, Isafjordsgatan 22, Kista.

Tryck: Universitetsservice US-AB, Stockholm 2020

To my family

Resumo

O advento dos sistemas multicore tem despertado grande interesse na Memória Transacional (MT). MT é um paradigma de programação paralela que coordena acessos concorrentes à memória partilhada, incorporando transações em linguagens de programação. A MT transfere a complexidade de coordenação e sincronização de *threads* dos programadores para o compilador, a implementação da MT ou até o hardware.

No entanto, as implementações existentes da MT não são compatíveis com uma abstração poderosa e intuitiva que é amplamente utilizada em ambientes modernos de programação paralela (por exemplo, C++, Java e JavaScript), ou seja, "futuros" (ou promessas). A abstração de futuro é bastante popular devido à sua capacidade de expressar de maneira intuitiva quer oportunidades de paralelismo quer dependências lógicas entre tarefas paralelas. Todavia, talvez surpreendentemente, o problema de como suportar a abstração de futuro numa implementação de MT ainda não tem sido estudado na literatura, embora os futuros representem um meio natural e conveniente para conseguir obter paralelismo intra-transações em transações de longa duração.

Esta dissertação visa preencher precisamente essa importante lacuna da literatura, investigando como reconciliar as abstrações de MT e a de futuros.

Esse problema é abordado introduzindo uma nova abstração, denominada futuro transacional, ou seja, uma transação executadas dentro de um futuro e gerada e avaliada por outras transações ou futuros transacionais. Esta dissertação define um conjunto de semânticas alternativas para as propriedades de atomicidade e isolamento de futuros transacionais, que visam explorar diferentes equilibrios entre facilidade de uso e eficiência.

Com base nessas semânticas, esta dissertação apresenta e avalia duas novas implementações de MT, que permitiram avaliar a eficiência da abstração de futuro transacional em sistemas reais.

Finalmente, esta dissertação aborda o problema de ajustar de forma automática o grau de paralelismo em sistemas de MT que suportam paralelismo intra-transacional. Esse objetivo é alcançado através da introdução de um sistema de aprendizagem automático que combina técnicas de buscas locais e orientadas por modelo (*Sequential Based Bayesian Optimization*), bem como mecanismos adaptativos de monitorização de desempenho.

Palavras-chave

Memória Transacional em Software, Programação Paralela, Futuros, Sincronização, Controle de Concorrência, Auto-ajuste, Otimização Bayesiana Seqeencial Baseada em Modelos.

Sammanfattning

Tillkomsten av flerkärniga datorsystem har väckt stort intresse för Transactional Memory (TM). TM är en paradigm för parallell programmering som samordnar samtidiga trådar genom att integrera transaktioner i programmeringsspråket. TM förskjuter bördan av att koordinera och synkronisera samtidiga trådar från programmeraren till kompilatorn, exekveringsmiljön eller till och med hårdvaran.

En kraftfull och intuitiv abstraktion som kallas futures används i flera moderna parallella programmeringsspråk, så som C++, Java och JavaScript. Denna abstraktion används i stor utsträckning på grund av dess förmåga att på ett naturligt sätt uttrycka möjligheter till parallellism och logiska beroenden mellan parallella uppgifter. Kanske överraskande så har problemet med att stödja future-abstraktionen i TM-implementationer inte studerats i litteraturen, även om den utgör ett naturligt och bekvämt sätt att möjliggöra parallell exekvering i långvariga transaktioner.

Avhandlingen syftar till att fylla just detta relevanta gap i litteraturen genom att undersöka hur man kan förena TM och futures.

Denna förening görs genom att införa en ny abstraktion, kallad transaktionell futures, d.v.s. transaktioner som utförs inslagna inuti futures och som skapas och utvärderas av andra transaktioner eller transaktionella futures.

Semantiken hos en transaktionell future beskriver de tillåtna beteendena med avseende på egenskaper som atomicitet och isolering. Flera semantiker defineras, och balansen mellan användarvänlighet och effektivitet utforskas för semantikerna. Baserat på dessa semantiker så presenterar denna avhandling två nya TM-implementationer. Effektiviteten hos den föreslagna transaktionell future-abstraktionen utvärderas i ett riktigt system som använder dessa TM-implementationer.

Slutligen behandlar denna avhandling problemet med att självjustera graden av parallellism i TM-system som stöder inkapslade parallella transak-

tioner inom transaktioner. Målet uppnås genom ett online-inlärningssystem som kombinerar modelldrivna (Sequential Based Bayesian Optimization) och lokala söktekniker, såväl som anpassade prestandaövervakningstekniker.

Nyckelord

Mjukvarubaserat transaktionellt minne, parallell programmering, futures, synkronisering, parallell exekvering, självjustering, sekventiell modellbaserad Bayesian-optimering

Abstract

The advent of multicore systems spurred great interest in Transactional Memory (TM). TM is a parallel programming paradigm that coordinates concurrent threads by incorporating transactions into programming languages. TM shifts the burden of coordinating and synchronizing concurrent threads from programmers to the compiler, run-time environment, or even hardware.

Unfortunately, though, current state-of-the-art TM implementations lack support for a powerful and intuitive abstraction that is widely used in modern parallel programming environments (e.g., C++, Java and JavaScript), namely "futures". The future abstraction is widely used due to its ability to express in a natural way opportunities for parallelism as well as logical dependencies among parallel tasks. Yet, perhaps surprisingly, the problem of how to support the future abstraction in a TM implementation has not been studied in the literature, although futures represent a natural and convenient means to enable intra-transaction parallelism in long-running transactions.

This dissertation aims at filling precisely this relevant gap in the literature by investigating how to reconcile the TM and the future abstractions.

This limitation is tackled by introducing a novel abstraction, called transactional futures,i.e., transactions that execute wrapped within futures and that are spawned and evaluated by other transactions or transactional futures.

The semantics of transactional futures describes the allowed behaviors with regards to properties such as atomicity and isolation. Multiple semantics are defined, and the trade-offs between ease of use and efficiency are explored for each. Based on these semantics, this dissertation presents two novel TM implementations. The efficiency of the proposed transactional futures abstraction is evaluated in a real system using the TM implementations.

Finally, this dissertation addresses the problem of self-tuning the parallelism degree in TM systems that support intra-transaction parallelism. This goal is achieved by presenting an online learning system that combines model-driven (Sequential Based Bayesian Optimization) and local searches techniques, as well as adaptive performance monitoring techniques.

Keywords

Software Transactional Memory, Parallel Programming, Futures, Synchronization, Concurrency Control, Self-tuning, Sequential Model-based Bayesian Optimization

Acknowledgments

First and foremost, I wish to express my deep gratitude to my advisors, Paolo Romano, and Seif Haridi. I am lucky to have Paolo as my advisor, who invested his time and energy into helping me to become an independent researcher. He has been a role model inspiring me to devote my passion to the work, especially when challenges and difficulties arise. This journey, which helped me grow as a researcher and a human being, will be a life-long treasure.

I appreciate having Seif as my advisor as his intellectual curiosity inspired me throughout the journey. Seif guided me into exploring various research and work opportunities, and those experiences nourished my research and presentation skills.

I would like to thank Luís Rodrigues for offering me the opportunity to study at Instituto Superior Técnico IST, Portugal. The research training and living experience in Lisbon has been one of the best moments during my Ph.D. study.

I would like to thank my co-authors, João Barreto, and Shady Isaa, for their effort and insight to improve my work.

Many other friends and colleagues also indirectly contributed to my work and made my doctoral journey a valuable experience. Over chats and course breaks, and perhaps other formal or casual occasions, Jim Dowling provided important encouragement and motivation through interesting open-ended research questions, opportunities, and insights. These drove me to pursue higher targets for myself and my work. Vladimir Vlassov, my second advisor, and Šarūnas Girdzijauskas, my quality assurance reviewer and doctoral study advisor always provided useful advice. Paris Carbone's talent and enthusiasm energized me after every hang-out and chat of ours.

I am grateful to my colleagues Amir Payberah, Fatemeh Rahimian, Cheng Li, Manuel Bravo, Zhongmiao Li, Lars Kroll, Alex Ormenisan, Amira

Soliman, Vasia Kalavri, Ying Liu, Hooman Peiro Sajjad, for their company and inspiring discussions during these years. Thanks to Daniel Castro for his effort in helping me with my Portuguese abstract. Special thanks go to Niklas Ekström for guiding me tirelessly when I started to conduct research and for helping me with the Swedish abstract.

Finally, I would like to thank the Erasmus Mundus Joint Doctorate in Distributed Computing program (funded by the EACEA of the European Commission under FPA 2012-0030), SSF funded E2E Clouds project for funding most of my doctoral research.

Last but not least, I am fortunate to have my parents' unrelenting support and encouragement. My dad taught me about determination and perseverance, and my mom instilled me with a sense of optimism, both of which allowed me to overcome difficulties, turning my doctoral studies into an enjoyable experience.

I am so fortunate to have my husband, partner, and best friend, Tian. Words cannot adequately express his unwavering support, patience, and care through this entire process. I am immensely blessed to have him in my life. My final blessing is found in our son, Yifei, who adds genuine joy to every single day.

Contents

Chapter 1

Introduction

The original 1965 Moore's law is an empirical observation that states that the density of transistors in integrated circuits doubles every two years [1]. This law has held during the following 30 years. During this period, processor got more efficient thanks to the ability of CPU manufacturers to integrate larger number of transistors, and, hence, support more complex features in hardware. Concurrently, as clock speed has kept on increasing exponentially, which combined with the constant reduction in the miniaturization scale of chips, lead around 2005 to hit critical thermal issues and cause clock speed to plateau approximately 3.5GHz (Intel Pentium 4). This led to a major paradigm shift at the computer architecture level, as most of the manufacturers have since then shifted their efforts from accelerating the performance of sequential processors towards the development of multi-core processors.

This paradigm shift had a great impact also for the software development process. If a software developer develops a program without taking into consideration the parallelism available at the hardware level, the performance of her programs will no longer benefit from the increase of computational capacity of future processors, which will be enabled by increasing the number of cores available on a single platform. Unfortunately, though, developing parallel applications is a notoriously complex task.

One of the most complex problems that needs to be tackled in order to build parallel applications that are not only correct, but also efficient and scalable, is how to synchronize the concurrent access to shared resources.

The conventional approach to solve the problem of synchronizing concur-

rent accesses to shared resources is to rely on the use of locks. Unfortunately, though, lock-based approaches suffer of several, well-known shortcomings. On the one hand, coarse-grained locks are easier to reason about, but suffer from scalability issues. Fine-grained locks, on the other hand, do scale better, but they raise a number of issues, such as dead-locks and live-locks, that can undermine the correctness of programs introducing subtle bugs that are notoriously hard to reproduce and fix. Moreover, locks are known to compromise a property that is regarded as essential in the design of software, namely composability. The simple example in Listing 1.1 illustrates that lock-based programming has inherent limitations and does not support the design principle of composability.

Listing 1.1: An example that illustrates the problem of lack of composability using locks. The withdraw method is correct when used in isolation. However, the transfer method, which reuses withdraw, can suffer of deadlocks, e.g., when two concurrent transfers A.transfer(B) and B.transfer(A) are executed.

```
Class Account{
    float balance;
    synchronized void withdraw(float amount){
        if(amount > balance)
            throw exception;
        balance−=amount;
    }

    synchronized void transfer(Account dest, float amount){
        this.withdraw(amount);
        dest.withdraw(−amount);
    }
}
```

Transactional Memory (TM) is probably among the abstractions for parallel programming that have garnered the largest interests of the research community over the last decades. TM relieves programmers from the complexity of defining lock-based inter-thread synchronization mechanisms to safeguard concurrent accesses to shared data. With TM, programmers need simply to define which code blocks should be executed as *transactions*. Under the hood, the TM run-time system automatically performs concurrency control ensuring that transactions only commit if their execution is equivalent to a serial one.

The ease of programming of TM is illustrated in Listing 1.2, which revisits the same code example considered in Listing 1.1, adapted to use TM instead of locks.

The TM abstraction is implementable in software [2], hardware [3], or combinations thereof, and its relevance has been strongly amplified by the recent integration of hardware TM support in the latest generations of CPUs by Intel and IBM, and by the inclusion of programming constructs for TM in the standard C/C++ languages.

Listing 1.2: The same code example of Listing 1.1, but using TM instead of locks as synchronization abstraction. Composability is preserved by the TM implementation, which is responsible for ensuring the correct execution of arbitrary complex code.

```
Class Account{
    float balance;
     void withdraw(float amount){
        Transaction.begin();
        if(amount > balance)
            throw exception;
        balance−=amount;
        Transaction.commit();
    }

     void transfer(Account dest, float amount){
        Transaction.begin();
        this.withdraw(amount);
        dest.withdraw(−amount);
        Transaction.commit();
    }
}
```

1.1 Problem Definition and Research Questions

Although today transactional memory has become a mature research field, with hundreds of papers published by academic researchers and industrial companies, the actual adoption of TM into mainstream programming strongly depends on how easy it is to integrate memory transactions into existing programming languages, abstractions, and patterns. Unfortunately, the current state-of-the-art implementations of TM are not compatible with important

programming constructs that are widely used in the programming community. One striking example of such a problem can be found with the powerful and widely-used abstraction of futures [4, 5]. Futures are a convenient programming construct that activates a parallel computation (typically encapsulated by a method in an object-oriented programming language) and returns a placeholder, called future (or sometimes promise [6]). The returned future can be used by the subsequent instructions on the calling thread (also called *continuation*) to check whether the computation has already completed and to obtain the result of the computation once it becomes available. Futures are part of C++, Java and the .Net platform (among others), hence a familiar construct to the average parallel developer in such environments.

Yet, perhaps surprisingly, futures are not compatible with any current state-of-the-art implementations of TM, although futures can be useful to enable intra-transaction parallelism for long-running transactions. Most of TM work assumes the sequential execution of operations in a transaction. However, there are applications that generate long transactions and for which it might be desirable to exploit parallelism within a transaction to improve performance.

Nevertheless, the intra-parallelism transaction was only studied assuming a fork-join parallel nesting model. Parallel nesting is one of the most well-known techniques. It allows multiple nested transactions to be executed in a dynamic extension of another transaction (often called parent transaction). Parallel nesting increases the expressiveness of TM as programmers are enabled to organize multiple parallel tasks to be executed atomically as a whole. Analogously to parallel nesting, futures allow programmers to express when parallelization is useful: at which point in an otherwise sequential program a parallel task should be started (i.e., when the future is created). Differently from the parallel nesting model, though, the future abstraction does not block the execution of the main thread, also called continuation in the context of futures, till the parallel sub-task it spawned completes execution. Further, futures allow greater flexibility in defining when the results of a parallel sub-task are actually required by the application: futures can be evaluated in an asynchronous fashion and in an order that is totally unrelated with the orders in which they were spawned, whereas parallel nesting abides by a fork-join model, which imposes a logical barrier that enforces the completion of all the sub-tasks.

Futures' integration with TM is promising to enhance the programmer's

control over computation patterns within a transaction. On the other hand, given the futures' ability to generate parallel computations with complex dependencies, it is far from trivial to define intuitive, yet precise, semantics for what concerns isolation and atomicity for a future and its continuation. In the context of TM, this work was never done to the best of our knowledge.

Another key problem that needs to be tackled by programmers who want to take advantage of intra-transaction parallelism is how to allocate resources between intra- and inter-transaction parallelism. The problem of identifying, or self-tuning the parallelism degree in TM systems has been long investigated for TMs not supporting intra-parallelism, but, to the best of our knowledge, has never been studied in the context of intra-parallelism-enabled-TM. Indeed, the problem complexity is inherently exacerbated, since these require to identify the optimal parallelism degree not only for top-level transactions but also for nested sub-transactions. The increase of the problem dimensionality raises new challenges (e.g., increase of the search space, and proneness to suffer from local maxima), which are unsatisfactorily addressed by self-tuning solutions conceived for flat nesting TMs.

In summary, this dissertation aims at addressing two main research questions.

1. How to integrate the future abstraction in a TM implementation?

   We decompose the question of how to integrate the future and TM abstraction in two (related) sub-questions.

   The first question is of theoretical nature and aims at formalizing which semantics should be enforced by a TM system that allows expressing intra-transaction parallelism via the future abstraction. The intuition at the basis of the tansactional future semantics is that a transactional future and its continuation should appear as mutually atomic. Yet, given the broad range of concurrency patterns that can be supported by futures, two main theoretical questions arise:

   a) How should the boundaries of a continuation be defined?
   b) Which serialization orders should be allowed between a transactional future and its continuation?

   The second research problem is of practical nature and aims at evaluating whether the proposed transactional future abstraction can be

implemented efficiently in a real TM system. More precisely, what overheads and what performance gains are generated by implementations of alternative theoretical models of transactional futures?

2. What is the optimal parallelism degree in TM systems that support intra-transactional parallelism, either through transactional future or parallel nesting?

   As mentioned earlier, a particularly challenging scenario for TM systems is when workloads contain a large portion of long-running transactions. Indeed, long running transactions suffer from long windows of vulnerability (i.e., the interval of time during which the transaction can be subject to conflicts with other transactions and abort), which makes them prone to prohibitively high abort rates. A possible approach to tackle the performance issues caused by long transactions is to reduce the number of concurrent transactions, while dividing these fewer transactions into smaller, nested sub-transactions that can then be executed in parallel on the available idle cores, thus, reducing the execution time and vulnerability window of the original top-level transactions. To fully exploit the potential of intra-transactional parallelism requires tackling a non-trivial problem that does not arise in TMs that do not support intra-transactional parallelism: identifying the right balance between inter-transaction and intra-transaction parallelism.

   Specifically, given a set n of available cores, in a intra-parallelism-enabled-TM system it is necessary to decide how many root(or top-level) transactions are allowed to be simultaneously active (t), and how many should be allocated to child (or nested) transactions (c) within each top-level transaction. In CPU-intensive applications, as it is typically the case for TM environments, one should choose t and c such that the system is not oversubscribed, (i.e., $t \times c \leq n$). In platforms where t and c can be sufficiently high, there will be many possible configurations that avoid over-subscription: the search space grows in fact quadratically with respect to the problem of tuning the parallelism degree in TMs not supporting intra-transactional parallelism, which has been subject of intense study in the literature [7–11]. Identifying the optimal configuration in the resulting bi-dimensional search space is far from trivial. Among the different configurations, it is necessary to

decide whether to favor inter-transaction parallelism while reducing intra-transaction parallelism, or vice-versa. This depends on complex, workload-dependent contention dynamics that arise both among top-level transactions as well as among the sub-transactions of the same (or different) top-level transactions; as well as on the overheads of spawning and maintaining nested transactions. Further, in contention-prone workloads, it is also not guaranteed that the optimal solution is the one that maximizes core utilization.

1.2 Thesis Contributions

Thesis: **This thesis advances the state of the art in the field of intra-transaction parallelism, by defining abstractions, algorithms and self-tuning mechanisms for taking advantage of intra-transaction parallelism in Transactional Memory.**

More precisely, this thesis fills an important gap in the TM literature by proposing a new powerful abstraction, the *transactional future*. Transactional futures, as the name suggests, combine TM with futures, by allowing programmers to exploit intra-transaction parallelism via the abstraction of futures, while delegating to TM the complexity of regulating concurrent access to shared data.

Referring to the research questions discussed in Section 1.1, in the following I summarize the main contributions of my thesis:

- **C1: Formalization of a set of semantics for transactional futures that explore different trade-offs between simplicity and efficiency.**

  Regarding the first question, to define the semantics of futures in a transactional context, this dissertation proposes several definitions that regulate different aspects of isolation and atomicity of transactional futures. As already mentioned the abstraction of transactional future allows a future to be submitted and evaluated by transactions. A transactional future is executed as a transaction that can access shared variables and finally return some result value. Intuitively, a future and its continuation should appear as mutually atomic.

  As a first step to formalize the atomicity between transactional futures, we have introduced the Future Serialization Graph (FSG). Via

the FSG, we can rigorously specify how the boundaries of a future's continuation should be defined, as well as establish the set of feasible serialization orders among transactional futures and their continuations. For what concerns the latter aspect, this dissertation formalize two alternative semantics that define plausible serialization order of transactional futures and their continuations: (1) Strongly Ordered Transactional Futures (SO), which requires that a transactional future is serialized before its continuation. (2) Weakly Ordered Transactional Futures (WO), which requires that a transactional future is serialized either before or after its continuation. While the SO semantics are simpler and, arguably, more intuitive than WO, WO allows the TM implementation to accept a larger number of concurrent executions histories. Thus, the WO semantic reduces the probability that transactional futures have to be aborted or blocked, but requires the programmers to reason on a lager set of possible inter-leavings between transactional futures and their continuations. This part of the work is discussed in detail in Chapter 3.

- **C2: Design and evaluation of TM implementations that support the proposed transactional future abstraction.**

  This dissertation presents the design and evaluation of two software-based implementations of TM that support different semantics for transactional futures. These implementations are thoroughly evaluated by means of an experimental study that includes both synthetic benchmarks and standard TM benchmarks, which were adapted to make use of transactional futures. Via this study, we shed light on the costs and benefits of the proposed semantics for transactional futures, by quantifying both the overhead introduced by the underlying concurrency control mechanism and the gains stemming from the ability to define multiple serialization points for transactional futures.

  The first implementation introduced by this dissertation is SO-JTF (Strongly Ordered Java Transactional Futures), a TM that provides support for transactional futures that abide by Strongly Ordered semantics, i.e., whose serialization order coincides with their submission point. SO-JTF manages the sub-transaction's metadata by means of tree which is used to determine the serialization order of transactional futures. The experimental results with SO-JTF show that the use of

futures allows not only to exploit parallelism within transactions, but also to reduce the cost of conflicts among top-level transactions in high contention workloads. This contribution is presented in Chapter 4.

The second TM implementation introduced by this dissertation is WO-JTF (Weakly Ordered Java Transactional Futures), a TM that ensures Weakly Ordered semantics, i.e., a transactional future can be serialized before or after its continuation. WO-JTF employs multi-versioning to regulate concurrency among top-level transactions. Further, WO-JTF maintains, for each top-level transaction, a graph that is used to track the logical dependencies developed among the sub-transactions. Our experimental analysis of WO-JTF allowed for quantifying both the additional overheads that one has to incur to support WO semantics, as well as the gains that such semantics enable in terms of reduction of conflict probability and avoidance of straggling effects, i.e., stalls that arise, with SO semantics, when a thread requests to commit a transactional future F but has to block until any transactional future $F\prime$, which has to be serialized before F, has not committed, yet. WO-JTF is presented in Chapter 5.

- **C3: Self-tuning the parallelism degree in TM systems that support intra-transaction parallelism.**

  This problem is addressed by introducing AUTOPN, which represents, to the best of our knowledge, the first system capable of automating the tuning of the parallelism degree in TMs that support intra-transaction parallelism (e.g., via parallel nesting or transactional futures).

  AUTOPN leverages an innovative design that combines model-driven and local-search techniques. In more detail, AUTOPN operates in an online fashion, i.e., it assumes no *a priori* knowledge on the workload to be optimized and explores at run-time a small number of configurations before outputting a recommendation. The online exploration of configurations is first driven by a regression model based on the lightweight M5P decision tree algorithm [12]. This model is used by a Sequential Model-based Bayesian Optimization process [13], which aims to: (1) identify the most promising configurations to explore next, and (2) determine when to stop exploring (stopping criterion). Finally, the model-driven exploration phase is complemented by a refinement

phase, using a local search based on a simple hill-climbing strategy that aims at increasing robustness against modeling errors.

We further address the problem of tuning the duration of the monitoring windows used to collect feedback on the system's performance. This problem is tackled by introducing novel, domain-specific, mechanisms that aim to strike an optimal trade-off between latency and accuracy of the self-tuning process. The proposed solution was evaluated via an extensive experimental study, whose results highlight gains of up to 45× in terms of increased accuracy and 4× faster convergence speed with respect to several online optimization techniques (gradient descent, simulated annealing and genetic algorithm), some of which were already successfully used in the context of TMs without support for intra-transaction parallelism

AUTOPN is detailed in Chapter 6.

1.3 List of Publications

The majority of the content of this dissertation is based on the following published work, which have been peer-reviewed. Some paragraphs of this dissertation contains verbatim quotes from these publications.

Workshop Paper

P1 Jingna Zeng, Paolo Romano, Luís Rodrigues, Seif Haridi, João Barreto (2015, July). In search of semantic models for reconciling futures and transactional memory. In 7th Workshop on the Theory of Transactional Memory (WTTM) .

Contribution: The author of this dissertation is the major contributor of the work. She has contributed to develop and formalize the abstraction of transactional future. She was also a major contributor to write the paper.

Conference Papers

P2 Jingna Zeng, João Barreto, Seif Haridi, Luís Rodrigues, Paolo Romano: (2016, August). The Future (s) of Transactional Memory. In

2016 45th International Conference on Parallel Processing (ICPP) (pp. 442-451). IEEE.

Contribution: The author of this dissertation is the major contributor of the work. She has contributed to develop the algorithms, optimized the implementation of the prototype and conducted its experimental study. She was also a major contributor to write the paper.

P3 Jingna Zeng, Paolo Romano, João Barreto, Luís Rodrigues, Seif Haridi: (2018, May). Online tuning of parallelism degree in parallel nesting transactional memory. In 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (pp. 474-483). IEEE.

Contribution: The author of this dissertation is the major contributor of the work. She has contributed to develop the algorithms and implemented prototype as well as conducted the experimental study. She was also a major contributor to write the paper.

P4 Jingna Zeng, Seif Haridi, Shady Issa, Luís Rodrigues, Paolo Romano: (2020, July). Brief Announcement: Giving Future(s) to Transactional Memory. In Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). ACM.

Contribution: The author of this dissertation is the major contributor of the work. She has contributed to develop and formalize the abstraction of transactional future. She was also a major contributor to write the paper.

P5 Jingna Zeng, Shady Issa, Seif Haridi, Luís Rodrigues,Paolo Romano: Investigating the Semantics of Futures in Transactional Memory Systems. Under submission.

Contribution: The author of this dissertation is the major contributor of the work. She has contributed to develop the algorithms, optimized the implementation of the prototype and conducted its experimental study. She was also a major contributor to write the paper.

The content of Chapter 3 is based on papers P1, P2, P4, and P5, while Chapter 4 is based on paper P2, and Chapter 5 is based on paper P5. Chapter 7 is based on paper P3.

1.4 Dissertation Outline

The structure of this dissertation is organized as follows.

Chapter 2 provides a background on the future abstraction and reviews the state of the art on the design, implementation and self-tuning of transactional memory systems.

Chapter 3 introduces and formalizes the semantics of transactional futures.

Chapter 4 and Chapter 5 present the design and evaluation of two alternative implementations of the transactional future semantics formalized in Chapter 3.

Next, Chapter 6 proposes an online self-tuning system to tackle the problem of jointly optimizing the degree of inter- and intra-transaction parallelism.

Finally, Chapter 7 concludes this dissertation and presents a series of directions for future research.

Chapter 2

Background

This chapter presents the background and the related work of this dissertation. Section 2.1 introduces the future abstraction. Section 2.2 reviews conventional synchronization mechanisms for concurrent programming. Section 2.3 introduces the TM abstraction and presents several safety and liveness guarantees adopted in TM systems. The implementation of several software-based and hardware-based TM systems is overviewed in Section 2.4 and Section 2.5, respectively. Next, Section 2.6 presents extensions of the TM abstraction to support parallel nesting semantics. Further, as discussed earlier, this dissertation proposes an online self-tuning mechanism to adjust the parallelism in TM. Therefore, Section 2.7 reviews existing research that tackled the problem of self-tuning different aspects and configuration parameters of a TM system.

2.1 The Future Abstraction

The abstraction of futures was first proposed by Halstead [4], as a synchronization and scheduling language mechanism. The motivation at the basis of futures is to exploit the observation that, in most of the languages, the return value of a function is often not actually needed for some time. This opens the possibility to exploit hidden parallelism. In Multilisp, the future construct serves as a primary method to execute and synchronize parallel tasks. The future abstraction is invoked using the syntax:

(**future** *(future expression)*)

This construct immediately returns a reference to a future object *f*. This object can be treated as a "future" or "promise" to deliver, and at the meantime, concurrently starts the execution of the *future expression*. Before the execution ends, the future is said to be *undetermined*. The future object reference *f* can be stored or passed to other functions. If any function needs the future expression's value *v*, it can call:

(**future-eval** *f*)

which returns *v* immediately, if the execution of *future expression* has completed, or it will block until *v* is ready. When the execution finished, the yielded value is associated with the future, which is now *determined*. If an operation, such as addition, uses a future construct as an operand, it will block until the future becomes *determined*. On the other hand, it is often the case that operations, such as assignment or parameter passing, do not need right away the value associated with the future expression and can be, thus, executed even in the presence of *undetermined* futures. In this case, the future abstraction effectively allows for unlocking parallelism in a program. Furthermore, the synchronization between the producer of the future and user of the future is implicit, which is an important characteristic to hide the synchronization details from the programmer. Nonetheless, to use Multilisp future, the programmer has to be careful to assign independent tasks between a future and its, so called, *continuation*, i.e., the task executed by the future producer up to the point where the future is needed. If a future and its continuation share any resource (e.g., access a shared memory region), additional synchronization mechanisms should be employed to preserve the application correctness from the occurrence of concurrency anomalies.

Nowadays, futures have reached a broad adoption among mainstream programmers for their ability to expose fine-grained parallelism in an intuitive fashion. As of the writing of this dissertation, there are at least dozens of programming languages that support futures, either directly as a first-class programming construct or via some standard library. In Java, for instance, futures have been introduced since 2004, with the Java 1.5 release. As shown in the example below, in Java, programmers create a closure that implements the “Callable” interface, then this closure is submitted to an executor service to return a future reference *f*. Whenever the result of the future is needed, programmers use the *get* method to retrieve the value, possibly waiting for it to become available in a transparent way

for the programmer.

```
/* Java Future example */

class Task implements Callable<T>{
    @Override
    public Integer call() throws Exception {
        execute an asynchronous task;
        return T;
    }
}

ExecutorService executor = Executors.newCachedThreadPool();
Future<T> future = executor.submit(new Task());

someOtherOperations();
useFuture(future.get());
```

Java futures [14] can be seen as a form of method-level parallelism, as they can be used to explore parallelism in programs by forking the execution flow transparently at method calls. However, current implementations of futures provided in the Java Development Kit are not concerned with regulating concurrent access to shared data among the asynchronous work performed by different futures and continuations. As mentioned above, additional synchronization mechanisms should be used to this end.

Safe futures [15] use techniques similar to those used in software-based thread-level speculation [16] to avoid pathological side-effects between a future and its continuation if they both access shared locations. With safe futures, even though some parts of the program are executed concurrently and may access shared data, the equivalence of serial execution is preserved. However, safe futures assume that the underlying program is single-threaded. In other words, that the only parallel threads consist of the future and its continuation. This constitutes a crucial hindrance for the adoption of safe futures in generic multi-threaded applications, which are parallelized using coarse-grained threads. To the best of our knowledge, this dissertation is the first to propose safe support for futures in the scope of multi-threaded programs.

Another interesting context for exploiting futures is in the context of concurrent shared data structures. Kogan and Herlihy [17] studied how futures can be used to optimize type-specific, long-lived concurrent data structures. Indeed, these authors had already discussed, as an interesting

future work direction, the possibility of combining the abstractions of futures and transactions. Our work explores this idea and proposes, to the best of our knowledge, the first generic solution (i.e., not restricted to shared data structures) that supports the execution of arbitrary code in futures that can be executed within the context of atomic transactions.

This section discussed the future as an abstraction to coordinate parallel tasks. Overall, futures are a powerful and intuitive mechanism to express logical dependencies among concurrent tasks that allows for retrieving each task's result at the point where it is needed. However, the future abstraction leaves unsolved the problem of how concurrent accesses to shared objects issued by futures and continuations should be regulated, requiring for additional, *ad-hoc* mechanisms.

2.2 Conventional Synchronization Mechanisms for Concurrent Programming

As already mentioned, since 2004 the computational capacity of CPUs has kept on growing mostly thanks to the advent of multi-core architectures. As a consequence of this architectural shift, parallel programming, once confined to the niche of scientific computing, has entered the mainstream software industry as an essential instrument to take advantage of the full performance potential of current multi-core processors.

One crucial problem that has to be appropriately tackled to ensure application's correctness in parallel programs is how to synchronize concurrent access to shared resources, such as shared memory regions.

Synchronization aims to ensure that every execution of the concurrent program is correct, in the sense that it complies with a consistency requirement. An consistency requirement example is serializability [18, 19], which, roughly speaking, ensures that the execution of a concurrent program has to be equivalent to that of a sequential program. Traditionally the way to implement synchronization is based on locks. In the following, different lock abstractions and their implementations are reviewed.

2.2.1 Mutex Locks

Dijkstra was the first to formalize the problem of synchronization in parallel programs in his seminar paper [20] in 1965. Assume there exists three code

segments, A, B, and C, which access some shared resources, for example, a data structure, network, or peripheral devices. Assume also that, at any point in time, this shared resource can be safely accessed by at most one code block. The code blocks A, B, and C are typically referred to as protected sections, or critical sections.

The problem of preventing several processes to access critical sections in parallel is called the Mutual Exclusion Problem. Mutual exclusion (abbreviated for mutex) is the most basic form of synchronization. Generally, mutex is a mechanism to prevent multiple processes from accessing critical sections in parallel. The implementation of a mutex requires the design of an *entry* and an *exit* protocol to enclose a critical section, so that any critical section is executed by only one process at a time. Let us denote the entry and exit protocol of a mutex as *acquire_mutex* and *release_mutex*, respectively. As such, any critical section code segment would be defined as below:

```
acquire_mutex()
    critical_section_code()
release_mutex()
```

If multiple processes try to execute *acquire_mutex*, only one process wins and the others will block waiting for the mutex to become available again.

A correct mutex implementation should respect two constraints: namely mutual exclusion and starvation freedom. These two properties corresponds to a problem's safety property and liveness property (The notions of safety and liveness were first introduced by L. Lamport in [21]). A safety property states that "*something bad will never happen*", or, alternatively, that some invariant encoding the application's correctness should never be violated in any execution. In this scenario, mutual exclusion ensures that only one process is able to execute a critical section. On the other hand, a liveness property states that "*something good will eventually happen*" and in the case of mutex implementation, it indicates that if a process requests to acquire a mutex, it will eventually be granted access to it.

The problem of how to solve the mutual exclusion problem, i.e., how to implement the mutex abstraction, has been studied in the literature thoroughly. Next, we will review some of the most popular approaches that rely either on hardware or on software.

Hardware-based implementations. Modern processors provide a set of hardware-backed atomic instructions that aim at facilitating the implemen-

tation of mutex objects. Here we review two of the most popular ones.

TestAndSet: The TestAndSet instruction sets a memory location to 1 and returns its previous value. As mentioned earlier, this operation is atomic, which means that, even in the presence of concurrent invocations of this primitive by different threads, the concurrent execution of these primitives is guaranteed (by the hardware) to be equivalent to a sequential execution. If multiple processes try to execute TestAndSet at the same time to acquire a lock, only one process can ever be granted access to it. The remaining processes must wait for the "winning" one to exit the critical section. An example of using TestAndSet primitive to implement mutex is presented below:

Listing 2.1: TestAndSet based mutual exclusion

```
int lock=0;
while(TestAndSet(lock)==1);
critical_section_code();
lock=0;
```

CompareAndSwap: The semantic of the CompareAndSwap(CAS) primitive semantic is to compare the current value of a memory location with a reference value passed as input, and if they coincide, the memory location will be set to store a new value, also passed as input to the CAS. In other words, a CAS is an atomic conditional write: if a concurrent process manages to update the memory location targeted by a CAS that is being executed by a second process, the CAS write of the latter process will fail. Whether CAS has successfully made the substitution is indicated by its return value, which is either a Boolean value or the previous value in the memory location.

Listing 2.2: CAS based mutual exclusion

```
int lock=0;
while(CAS(lock,0,1)==1);
critical_section_code()
lock=0;
```

Software-based implementations. The mutex abstraction can be implemented also without the availability of atomic operations at the hardware level. Below, we overview two well-known approaches that operate fully in software and whose correctness does not hinge on the availability of atomic

operations.

Lamport's Bakery Algorithm: This algorithm [22] was the first to solve the mutex problem using non-atomic objects. The design is inspired by a queuing ticket machine. The machine gives each customer a unique ticket number at a baker's store, and the customs will be served based on the ordering of the number that they obtained.

If a process p wants to enter a critical section, p received a number n, which identifies p's priority, and processes enter their critical sections according to the priority. Due to the absence of atomic objects, two processes may obtain the same number n. In this case, the process's identifier i is used to break ties and establish a total order between all processes competing for critical section. In this algorithm, processes are served per their order of request. Thus, this approach provides a First-In-First-Out(FIFO) fairness.

One of the limitations of Bakery's algorithm is that the timestamps used to establish access to critical section can grow in an unbounded way. This issue is addressed by the BLRU (Bounded Least Recently Used) algorithm by Aravind [23]. BLRU ensures a different fairness property than that of Lamport's bakery algorithm. Unlike the Bakery algorithm, which calculates the timestamp of a process before it enters the critical section, i.e., at the entry section, BLRU algorithm calculates the timestamp of a process at its exit section. To avoid timestamps to grow unbounded in size, BLRU adds a conditional check upon exit section: when a timestamp reaches the maximum number N, every process' timestamp will be reset to its initial value. It can be shown that BLRU does not provide FIFO semantics, unlike the Bakery's algorithm, but rather an alternative fairness property known as Bounded Least Recently Used.

Advantages and limitations. The Mutex abstraction and its implementation have been long studied in the literature. Decades of research in this area and the availability of dedicated hardware supports have led to the availability of very efficient implementations.

Unfortunately, though, developing lock based synchronization schemes that are both efficient and correct is notoriously known to be a complex task for application programmers.

One of the key problems is inherently related to the choice of the proper granularity with which locks are logically assigned to memory regions or

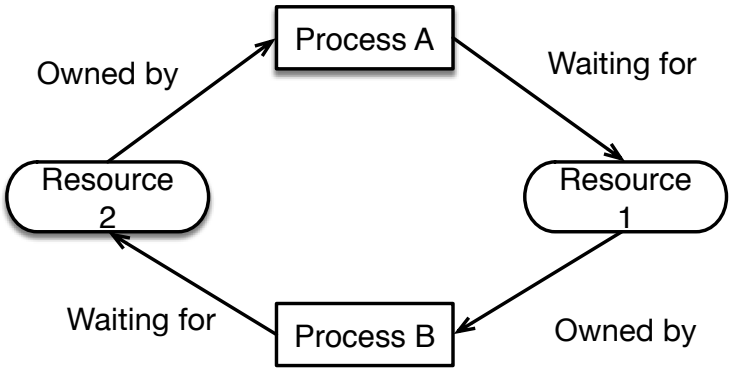


Figure 2.1: A Deadlock Example.

objects. Let us consider a concurrent queue as an example. An approach that uses a single coarse-grained lock to protect the whole queue is easy to reason about. Unfortunately, though, such an approach is far from being a scalable one, as it prevents any concurrent access to the queue.

On the other hand, fine-grained locking schemes can achieve scalability, but this comes at the cost of simplicity: if every entry of a concurrent queue is protected by a corresponding lock, operations that access different positions in the queue can in theory be executed in parallel. However, whenever threads need to acquire multiple locks to ensure atomicity, the program becomes susceptible to deadlocks and livelocks [24], which are notoriously hard to reason about and debug. Figure 2.1 demonstrate a simple deadlock scenario: process A has already acquired a lock on resource 1, then it requests a lock for resource 2, on the other hand, process B has already acquired the lock on resource 2 and then requests a lock on resource 1. Both processes are blocked, and none can make progress. Another key drawback of locks, which was already mentioned in Chapter 1 is that its usage can lead to break a property that is regarded as fundamental for modern software, i.e., composability [25].

2.2.2 Read Write locks

A Read Write Lock (RWL) extends the mutex abstraction by distinguishing whether processes that require access to the critical section into two classes: readers and writers [26]. A RWL allows an arbitrary number of readers to

access the critical section, provided that no writer has access to it; writers, conversely, requires exclusive access, locking out any other reader or writer.

Implementations. Nowadays, RWLs are supported by a large number of programming languages, e.g., in C/C++ via the POSIX standard *pthread* library, and in Java's standard library via the *ReadWriteLock* interface. In the literature, there are several proposals to efficiently implement RWLs [27]. Some of them assume the pre-existence of simpler synchronization primitives, such as mutexes or conditional variable, etc. Below, we show an example implementation based on two mutexes. The idea is to use a counter *reader_count* to track the number of concurrent readers. Furter, one mutex *reader* is used to guard the counter, and a second mutex, *writer*, is used to ensure mutual exclusion among writers.

Listing 2.3: Mutex Based Read Write Lock

```
Read_Begin
{
    aquire_mutex(reader);
    reader_count++;
    if(reader_count==1)
            aquire_mutex(writer);
    release_mutex(reader);
}

Read_End
{
    aquire_mutex(reader);
    reader_count--;
    if(reader_count==0)
            release_mutex(writer);
    release_mutex(reader);
}

Write_Begin
{
    aquire_mutex(writer);
}

Write_End
{
    release_mutex(writer);
}
```

An alternative implementation of RWL is based on the, so called, Read-copy update (RCU) [28] technique. RCU is a synchronization mechanism that allows multiple reads to operate concurrently with a single writer. The basic idea behind RCU is to defer the update until after the pre-existing reads finish while applying the write to a copy of the data. RCU is beneficial for read-intensive workloads and is immune from read-side deadlocks, because a reader can never prevent a writer from making progress. However, RCU writes are costly, since a new copy of the object has to be produced whenever a writer is granted access to the critical section. A number of alternative RCU implementations have been proposed in the literature [29–31] as the design of an RCU implementation implies a complex trade-off: RCU aims at enhancing the efficiency of readers, and as a result, the writer's synchronization logic becomes more complex.

Advantages and limitations. As already mentioned, RWLs enhance parallelism for readers, by ensuring that multiple readers can read data in parallel. In the meanwhile, RWLs impose higher cost for writers compared to mutexes. As such, RWL are adopted typically in read-intensive scenarios.

Despite allowing more concurrency for readers, RWLs suffer of the same drawbacks of mutexes for what concerns ease of programming. In fact, just like mutexes, RWLs can suffer from issues like deadlocks and livelocks. In terms of simplicity of use, actually RWLs can arguably be considered even more complex than mutexes, as they place a heavier burden for programmers, which in this case are faced with the additional problem of having to distinguish readers from writers.

2.3 The TM Abstraction

Just like high level programming languages free developers from having to write their programs in assembly and analogously to automatic garbage collection, which free programmers from having to worry about dynamic memory management, the TM abstraction can be seen as a step towards effortless concurrent programming.

TM [2, 3] borrows the abstraction of atomic transaction from the database literature and applies it as a first-class abstraction in the context of generic (i.e., not sand-boxed) parallel programs. By requiring programmers only to identify which code blocks should be executed atomically, and not how atomicity should be achieved, TM has been shown to effectively simplify

the development of concurrent applications [32–36], while delivering performance on par with (and sometimes even higher than) complex, hand-crafted fine-grained locking mechanisms [37].

The assumed system model on which a TM system is built is an asynchronous multiprocessor system. Each process, or thread, in the system executes a sequential program and a number of threads can be executed in parallel. Threads, denoted T_1,....,T_m communicate by accessing to shared objects. A shared object exposes two primitive operations, i.e., *read* and *write*. A $read(x)$ operation returns the associated value to the shared object x, while a $write(x, v)$ operation can be used to store value v in the shared object x.

A transaction is a sequence of read and write operations, whose start is demarcated by a *begin* operation. The end of a transaction is demarcated by the request to *commit* or *abort* a transaction, which makes all the transactions' effects (i.e., updates) visible to the other threads in the system or cancel all of its effects, respectively. An abort event can also be raised by the underlying TM system while a thread is executing a transaction, which also leading to cancelling all of the transactions' effects.

Below is a code example that illustrates the usage of TM to transparently synchronize the execution of the *transfer_money* method, which transfers money from a source to a target bank account, while ensuring that the balance of any bank account can never become negative.

The responsibility for ensuring the correct execution of transactions is delegated to the TM library, which can be implemented in software, hardware or via hybrid approaches that combine both software and hardware techniques.

Listing 2.4: A bank example using transaction

```
void transfer_money(Account to_account, Account from_account, int amount ){
        begin();
        if(read(from_account.balance) < amount){
                        abort();
                }
        write(to_account, read(to_account) + amount);
        write(from_account, read(from_account) − amount);
        commit();
}
```

Below an overview of the correctness properties that are typically guar-

anteed in a TM system is provided. Roughly speaking, the TM abstraction guarantees that the concurrent execution of a set of transactions can be explained via some equivalent sequential execution of the same set of transactions. More formally, the guarantees provided by a TM system can be classified into safety and liveness properties. Next, each of these two classes of properties is overviewed.

2.3.1 Safety Properties

A safety property states that a system can never enter an erroneous state. The safety properties for TM are rooted in the safety properties originally introduced in the database community, and thus we will start with serializability.

Serializability and Strict serializability. TM shares its fundamental abstraction, i.e., transactions, with database systems. So, it is unsurprising that the consistency criteria originally proposed in the database literature share commonalities with the ones used in TM systems.

In the database literature, serializability is probably the most well-known consistency criterion. Serializablity [18] requires that the execution of the set of *committed* transactions can be explained by replaying sequentially those transactions in some order.

Strict serializability [38] represents a natural extension of serializability, by imposing an additional constraint that has to be satisfied by the equivalent sequential execution used to justify the original, concurrent execution: if a transaction T_1 is committed in real-time before transaction T_2 started, then T_1 must be executed before T_2 also in the equivalent sequential execution. In other words, strict serializability additionally requires that real-time ordering relations between the transaction in the original execution must be preserved in the equivalent sequential execution.

Opacity. Both serializability and strict serializability dictate rules that restrict the set of possible behaviours only for committed transaction. The reason behind is that, in a typical DBMS (DataBase Management System), transactions execute in a sand-boxed environment (i.e., the database engine), which restricts the set of possible anomalous behaviours that can be produced when a transaction observes an inconsistent state of the database

and, thus, is eventually aborted (e.g., in such a case the database engine will not crash or hang).

Conversely, the TM abstraction is used to regulate the execution of generic concurrent programs, which execute in non-sand-boxed environments. In such a case, if a transaction observes an arbitrarily inconsistent state, which breaks some application's invariants, the application can produce anomalous executions (e.g., generating division by zero exceptions or getting stuck in infinite loops), which can cause the application to crash or hang.

For this reason, the TM literature has proposed stronger consistency condition than (strict) serializability, which aim precisely at specifying the set of acceptable execution also for aborted transactions. The more popular safety property for TM is probably *opacity* [39, 40]. Roughly speaking, in addition to the guarantees already ensured by strict serializability, opacity requires that every transaction, including the ones that abort, observe a state that was produced by a serial history that contains only committed transactions.

More recently, other researchers [41, 42] have proposed alternative consistency properties, for example, Strong Transactional Memory Specification, which were specified by using the observational refinement technique [41]. Yet the intuition at the basis of these alternative definitions remains the same, i.e., provide intuitive semantics to programmers and prohibit concurrency anomalies from affecting the execution not only of committed transaction but also of transactions that have to be aborted eventually.

2.3.2 Liveness Properties

A natural starting point to specify progress guarantees for TM is represented by the liveness properties, the ones that were original proposed in the general context of concurrent programming, namely wait-freedom, lock-freedom and obstruction-freedom.

Wait-freedom guarantees that any thread can complete any operation in a finite number of steps [43]. This is a very strong property, as it requires a thread to make progress no matter what other threads are doing and regardless of other threads' speed of execution. *Lock-freedom* specifies that, if the program threads are run for a sufficiently long time, at least one of the threads makes progress. Lock-freedom is weaker than wait-freedom because it allows executions in which a single thread completes its opera-

tions, while all the remaining threads starve, i.e., complete no operations. Lock-freedom implementation usually involves some helping mechanism, so that if one thread can not make progress on its own, it will help another thread. *Obstruction-freedom* [44] is the weakest among these three progress guarantees, as it only ensures progress if the threads execute in isolation, i.e., other threads do not run at the same time.

Existing liveness properties for TM can be seen as a specialization of these "generic" progress guarantess for concurrent systems. Specifically, in the context of TM, the natural notion of progress corresponds to the ability for a thread that runs infinitely long to commit an infinite number of transactions. Adopting this definition of progress, Bushkov and Guerraoui in [45] proposed the following liveness properties for a TM implementation:

- *Local Progress*, which requires that *every* thread that runs infinitely long commits an infinite number of transactions.
- *Global Progress*, which requires that *some* thread that runs infinitely long commits an infinite number of transactions.
- *Solo Progress*, which requires that any thread that runs *alone* (i.e., in absence of concurrency) infinitely long commits an infinite number of transactions.

An additional example of liveness properties for TM is progressiveness [46], which focuses on specifying the conditions under which a transaction is allowed to abort. Specifically, progressiveness requires that a transaction aborts only because of data conflicts with a concurrent one, i.e., when they are both trying to access the same data item and at least one of the transactions is trying to update it.

A related property is permissiveness [47], which, intuitively, specifies that a TM implementation should never generate an abort. More precisely, various levels of permissiveness have been defined. These include: *single-version permissiveness* [47] assumes a single-versioned TM implementation and allows for many spurious aborts; *MV-permissiveness* [48], which allows a transaction to abort only if it is an update transaction that conflicts with another update transaction (i.e., read-only transactions never abort and do not cause aborts of update transactions); *online permissiveness* [49] rules out *any* abort, which can only be achieved using extremely complex (and, thus, prohibitively costly) TM implementations.

2.4 Software TM Implementations

This section overviews the main choices that affect the design of a software-based TM (STM) (Section 2.4.1) and then presents some representative STM implementations (Section 2.4.2).

2.4.1 STM Design Space

The design space of STMs is pretty vast, which explains the abundance of diverse STM systems presented so far in the literature. In the following, I discuss some of the main design choices underlying the implementation of an STM system. This analysis will be useful also to clarify the differences among the STM systems presented in Section 2.4.2.

Object versus Word Based STM

This design choice affects how does the STM views the (transactional) heap space. Word-based STMs access to the data words directly while object-based STMs use an additional level of abstraction and access every data via an object.

Timing of Conflict Detection

Conflicts in a TM arise when two transactions access the same data, and at least one of them writes the data. A key design choice for a (S)TM system is when to detect conflicts, given that conflict detection introduces costs, but postponing it exposes the transaction to the risk to have to roll-back at a later stage, leading to wasting work.

One option is to detect the conflict *eagerly*, whenever a (read or write) operation is executed. An opposite possibility is to detect conflicts *lazily* or as late as possible, i.e., when a transaction tries to commit. A *mixed* conflict detection policy is to detect write-write conflict eagerly and read-write conflict lazily. The reason is that when write-write conflict happens, at least one of the transaction must abort, therefore it is beneficial to detect such conflict as soon as possible. On the other hand, when read-write conflict happens, there is chance for both transactions to commit, as long as the read happens after the write. The choice of the optimal approach is workload-dependent [50]. As a general rule, if conflicts are common, eager conflict detection policies tend to outperform lazy approaches, and vice versa.

Granularity of Conflict Detection

The granularity at which conflicts are tracked by an STM can also have a key impact on its practical efficiency. Fine-grained tracking schemes, unlike coarse-grained ones, are spared from *false conflicts*. As such, fine-grained approaches spare transactions from spurious aborts that may arise with coarse-grained approaches due to access aliasing problems. However, fine-grained conflict detection schemes come at the cost of maintaining a larger number of metadata.

This problem is particularly relevant in word-based STMs, where detecting conflicts at the level of an individual (32- or 64-bits wide) memory word requires storing for each word in the transactional heap some meta-data (also known as Ownership Records [2]) of relatively large size (e.g., a lock and often additional timestamps). Thus, when fine-grained tracking is used with large transactional heaps, the space overhead due to transactional metadata can be prohibitive large. Further, scattering the transactional metadata over a large address space leads to poor access locality, hindering cache efficiency, which translates in larger overheads for the STM layer.

Update Policy

The update policy defines the mechanism a TM system employs to manage its updates to shared data. One possible approach is *eager update* or *direct update*, where a transaction's write is applied directly to the shared data in memory (typically after having acquired a lock on that data). As a transaction may still abort subsequently, each transaction maintains a *undo-log* which stores the old value overwritten by the transaction. In the case of abort, the transaction is responsible for restoring the old value by replaying the values stored in its undo-log.

In contrast, another mechanism, called *lazy update* or *deferred update*, defers applying the updates until the commit phase. As such, a transaction maintains, throughout its execution, its tentative writes in a private write-set, also known as a *redo-log*. Consequently, the transaction's reads need to consult its write-set, to retrieve any value previously written by the same transaction. Lazy update naturally prevents another concurrent transaction to obverse any tentative/uncommitted write. If a transaction aborts subsequently, the write-set is being discarded so as the transaction holding it.

2.4.2 STM Implementations

As already discussed, an STM implementation can choose various techniques in a broad design space. In the following, we overview several state-of-art STM systems, which opt for different choice in the design space that we have defined in the previous section.

TL2(Transactional Locking 2). TL2 [51] is a word-based STM system that ensures opacity. TL2 uses a single global clock to track the progress of the system. In other word, this logical clock is used to count the number of committed update transactions. Naturally, the clock is used to timestamp each shared data as well, to ensure that each transaction reads a consistent data according to their assigned timestamp. When a transaction starts, it reads the current timestamp value. During its execution, a transaction maintains its read-set and write-set. A transaction reads by first consulting its write set, to ensure consistency in presence of earlier writes. If this is not the case, a local copy is created and initialized to the value read from the shared variable. A read operation is *invisible*, as it does not alter the state of any shared variable. A transaction write does not write to the shared memory directly, but to a local copy in the transaction's write set. Each shared variable is associated with two field, in addition to its value: a *timestamp*, which indicates the last transaction that updated this location; and an *ownership record*, which serves as a versioned write-lock for an actively writing location. From an outsider observer's point of view, a transaction is executed and committed at the logical clock's time point when the transaction gets started. When a transaction reaches its commit phase, TL2 uses a two-phase locking. It first acquires the locks, i.e., ownership records, of all the shared variables in the read set as well as the write set. Then the transaction has first to validate its read set, to make sure no transaction has committed and invalidated the transaction's read set during its execution. If this validation fails, the transaction aborts. If the validation succeeds, the transaction writes back the values according to the local write set, and release all the locks.

JVSTM (Java STM). JVSTM is an object-based STM system. It is the first TM implementation that uses a version-based scheme [52]. JVSTM provides opacity. It is a Java-based multi-versioned STM implementation. JVSTM introduces the abstraction of *VBox* (Versioned Box), which serves

as a container to store several versions of a transactional object. As illustrated in Figure 2.2, a *VBox* maintains a pointer to the head of a permanent list of versions, where each version in the list is established by a committed top-level transaction and has a value and a timestamp associated with it. *VBox* also maintains a pointer to the head of a tentative list of versions, where each item in the list belongs to a active transaction and thus a part of the transaction's write-set. Moreover, each tentative write points to an ownership record (orec) that encapsulates below information: the transaction that owns it, the transaction's status and the write's version. JVSTM is particularly relevant for my work presented in this dissertation, since the STM implementations presented in Chapter 4 and Chapter 5 used JVSTM as base STM and extended it to support transactional futures (with different semantics).

Similar to TL2, JVSTM also uses a global logical clock that is incremented along with transaction's commit. Upon the start of a transaction, the current global clock is read and used to establish the snapshot (i.e., the correct version) of data that the transaction should observe.

A transaction keeps a local write set and read set, which function similarly as in TL2 design. During execution, transactions write are buffered in a private write set. A transaction reads by first checking whether there is a local value available for the target data item, or VBox, in the transaction's write-set. If this is not the case, the transaction accesses the VBox's version list and searches from the head of list until a compatible version is met, i.e., the version with the largest timestamp that is smaller or equal to the transaction's snapshot timestamp. Through the use of multi-versioning, with JVSTM, read-only transactions can never abort or block because they can always read proper versions of shared variables. At commit phase, JVSTM employs a lock-free algorithm, which uses a helping mechanism to implement the following two steps in a non-blocking, yet atomic, fashion: increasing the global counter and writing-back the values from the transaction's write-set to the corresponding VBoxes.
Moreover, JVSTM provides a garbage collection mechanism to discard old versions that are no longer accessible by any active future transaction.

JVSTM uses a lock-free design for the commit phase. Every committing transaction enters a queue using non-blocking atomic operation (Compare-And-Swap), and the order in the queue establishes the serialization order of these transactions. If a committing transaction has not reached its turn to commit yet, the thread executing it will help the transaction in the head of

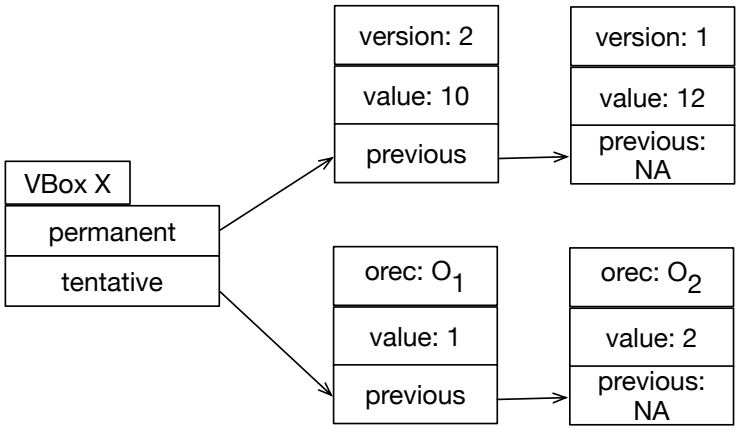


Figure 2.2: VBox Structure: A list of versions associated with a transactional variable.

the queue to validate and commit first.

NOrec. NOrec STM [53] does not maintain an ownership record for each transactional variable, hence the name No + Orec. NOrec provides opacity. NOrec transaction's writes are made using a local write-set and will write back at commit phase. NOrec is a system with global metadata; thus, it is designed to minimize the instrumentation overhead. Below, we emphasize the differences with other STMs; namely, its global metadata, its read procedure, and the value-based validation.

NOrec uses a global versioned lock. This lock is only acquired when an update transaction tries to commit and write back updated values from its local write set. The lock also serves as a logical clock. A transaction's timestamp is determined as the version number of this global lock when the transaction begins. The value of the lock gets incremented when a transaction gets committed and updated shared memory. As such, when a transaction detects that the global lock gets increased, it becomes aware that some concurrent transaction has updated some shared data item, and it validates the local read-set to detect write-after-read conflicts with concurrent committed transaction.

Specifically, the validation of a transaction's read set ensures that the

transaction is still valid because it means that the transaction is still serializable at this point in time. This validation is done by comparing the values of the read-set variable with the value in the shared memory. This read-set validation is performed again at the commit phase but with the global lock acquired.

TinySTM. TinySTM is a word-based STM implementation [54] based on LSA, i.e., Lazy Snapshot Algorithm [55]. TinySTM uses a global-version clock as TL2 does. TinySTM used eager version management as encounter-time locking (also known as eager locking, where the locks are acquired eagerly at the time of the first write operation by the transaction) is used instead of commit-time locking (also known as lazy locking, where the locks are acquired lazily at the time when the transaction is ready to commit). As specified by the LSA, if a transaction reads a shared variable whose ownership record's timestamp is larger than the transaction's timestamp, the transaction does not abort yet (unlike, e.g., in NOrec or JVSTM). The transaction will try to "extend" its snapshot by checking whether its local read-set is still up-to-date. If this is true, the transaction will set its timestamp to the current global-version clock value, hence logically extending its snapshot.

Due to its use of ownership-records (whose granularity is automatically adjusted using a self-tuning mechanism), TinySTM design imposes larger instrumentation overheads when compared to NOrec. However, precisely thanks to the use of fine-grained ownership-records, TinySTM spares many of the unnecessary validations that are triggered in NOrec whenever a concurrent transaction commits, which is arguably the main scalability bottleneck of NOrec.

We conclude this section by presenting Table 2.1, in which we summarize and aid the comparison of the main design choices underlying the different STM implementations presented in this section.

2.5 Hardware TM Implementations

As of the year 2012, commodity CPUs from mainstream manufacturers, such as Intel [34] and IBM [56–58], started to include support for hardware transactions.

STM implementations	TL2	JVSTM	Norec	TinySTM
Object vs Word	Word	Object	Word	Word
Timing of Conflict Detection	Mixed	Mixed	Lazy	Mixed
Granularity of Detection	Word	Object	Word	Word
Update Policy	Deferred	Deferred	Deferred	Both

Table 2.1: Summary of the main design choices underlying the STM Implementations overviewed in Section 2.4.2

To support the execution of transactions at the hardware level, the instruction set of these processor has been extended with *ad hoc* instructions to demarcate the start and end (i.e., commit or abort) of transactions. An example is represented by RTM (Restricted Transactional Memory) interface, which Intel introduced along with its HTM implementation (named Transaction Synchronization Extension or TSX) [59].

Although existing HTM designs differ in several aspects, all the currently available HTM implementations share a common trait: they implement the TM abstraction by extending the pre-existing cache coherence protocol. More in detail, processor caches are used to keep track of both the transactions' read-sets and write-sets. Writes are buffered and made globally visible at commit time, using a deferred update policy.

The design choice of relying on caches for storing transactional metadata causes existing HTM implementations to have a *best effort* nature. Unlike STMs, which only abort due to conflicts or explicit requests by the application, in HTM causes of "spurious" aborts include overflowing the cache capacity (i.e., the transaction accesses too many memory positions and its read-set and/or write-set exceed the cache capacity), but also any other event that causes the cache to be purged, e.g., system calls and interrupts.

Because of its best-effort nature, HTM normally requires the definition of a fallback execution path, which typically is a single global lock (SGL). The SGL is acquired whenever a hardware transaction exhausts its budget of attempts (either statically fixed or determined in an adaptive fashion [60]). The SGL is also subscribed (i.e., read) by every transaction running in the HTM system, which guarantees that whenever some transaction activates

Processor Type	IBM Blue Gene/Q	IBM zEnterprise EC12	Intel Core i7-4770	IBM POWER8
Conflict-detection Granularity	8-128 bytes	256 bytes	64 bytes	128 bytes
Transactional load capacity	20 MB	1 MB	4 MB	8 KB
Transactional store capacity	20 MB	8 KB	22 KB	8 KB
Roll-back only transactions	✘	✘	✘	✔
Suspend and Resume	✘	✘	✘	✔
Constrained Transactions	✘	✔	✘	✘
Abort Reasons	-	14	6	11

Table 2.2: HTM Implementations of Blue Gene/Q, zEC12, Intel Core i7-4770, and POWER8. Most of the data in the table was originally presented in [61].

the fallback path (i.e., acquires the SGL) any concurrent transaction executing in the hardware is forcibly aborted.

Table 2.2 provides an overview of four HTM implementations, three by IBM and one by Intel, highlighting some key commonalities and differences. The table analyzes the considered HTM systems according to the following characteristics:

- **Conflict detection granularity.** Current HTM implementations, as already mentioned, rely on the cache coherence protocol to detect conflicts and, as such, the granularity of conflict detection is typically at the level of an individual cache line. As shown in Table 2.2, different architectures use different cache line sizes. This can make existing HTM implementations more or less prone to suffer from spurious conflicts due to address aliasing problems.

- **Transactional capacity for load and stores.** Existing HTM implementations were integrated in processors with caches of different

sizes. Further, different HTM implementations track read-sets and writes differently in different levels of caches. This explain why their transactional capacity vary significantly. It can also be noted that the transactional load capacity is sometimes (e.g., in the Intel implementation) larger than the store capacity. This is quite useful, since reads are typically more frequent than writes in realistic applications and this is achieved by tracking the read-set in lower level caches that the ones used to maintain the write-set and possibly adopting space-efficient and probabilistic representations such as Bloom filters [62]

- **Rollback-only transactions.** A rollback-only transaction (ROTs) is a special type of transaction supported in the IBM POWER architecture since POWER8. Similarly to plain transactions, ROTs allow to demarcate a code block, whose effects are made atomically (all-or-nothing) visible to other concurrent transactions. ROTs, though, do not guarantee isolation between transactions, since the hardware does not keep track of transactions' read-sets. As a matter fact, ROTs were not designed to regulate concurrent access to shared-memory, but rather to ensure failure-atomicity semantics. Recently, though, several works have extended ROTs with thin software layers, yielding hybrid hardware and software TM implementations that guarantee different consistency semantics, such as opacity [63] and snapshot isolation [64, 65]

- **Suspend and Resume.** Suspend and Resume (S/R) is another feature introduced by IBM in its POWER architecture, along with ROTs. S/R enables a transaction to request its suspension and later resume execution. While suspended, all the memory accesses escape the transactional context, i.e., are treated as non-transactional. This feature allows transactions to externalize (part of) their updates before committing. This can be useful for a number of reasons, including debugging, as well as to enable the implementation of different types of hybrid TM systems [63, 66, 67].

- **Constrained transactions.** The IBM Blue Gene/Q and EC12 provide supports for a specialize transaction type, for which the hardware can guarantee successful execution, thus lifting the need to define a software fallback. To this end, though, the transactions need to respect several constraints, such as executing a limited number of instructions

(2 per IBM's ZTM [58]) and having a limited memory footprint (32 bytes per IBM's zTM [58]).

- **Abort Reason Code.** Another aspect that differentiates existing HTM implementations is their ability to report information on the cause that triggered a transactional abort — which, as already mentioned, in HTM include capacity exceptions and interrupts, among others. The abort reason is not only helpful for debug, at development time, but can also be useful at run-time, to provide an indication on whether it is worth to re-try a transaction that aborted or not [60].

To conclude, due to its hardware nature, HTM can spare the software instrumentation costs that are instead unavoidable in STM systems. However, due to their best effort nature, HTM is only effective with a limited number of workloads (e.g., short transactions that read and write a relatively small number of memory positions). As such, existing HTM implementations cannot be regarded as a universal,or general-purpose, synchronization mechanism that can cope (efficiently) with arbitrary workloads. In this sense, STM systems are not only more flexible and easier to extend (due to their software nature), but they also tend to provide a more robust performances in the presence of generic workloads.

For the above reasons, the TM implementations with supports for transactional futures, which will be presented in Chapter 4 and Chapter 5, have been developed using a pure software-based approach.

2.6 Transactional Nesting

This section focuses on analyzing the semantics of TM in presence of nested transactions, i.e. transactions whose execution is contained in another transaction. In the literature, several models of transactional nesting have been considered. Below we overview the main models for transactional nesting, classifying them according to two dimensions: i) the semantics of commit and abort events for nested transactions and, ii) whether nested transactions can execute concurrently or in a sequential fashion.

Committing and aborting nested transactions

A few main models have been proposed for defining the semantics of committing and aborted nested transactions, namely flat nesting , closed nesting,

and open nesting. Each of them is discussed next.

Flat nesting. The simplest way to support nesting is flat nesting, in which nested transactions are "flattened" and seen as an integral part of their outer transaction. In this model, the abort of an nested transaction will cause the abort of its outermost transaction. On the other hand, the commit of a nested transaction makes its write-set visible to its outer transaction; further, the read-set of its outermost transaction is logically extended with the read-set of the nested transaction, whenever this commits.

Flat nesting is attractive mostly due to its simplicity: a straightforward implementation of flat nesting is to associate a counter to track the nesting depth; the counter is incremented when one inner transaction is created and decreased when an inner transaction is committed. The outermost transaction is committed when the counter reaches zero. However, it is also inefficient because the abort of a nested transaction forces to roll back to the beginning of the outermost transaction.

Closed nesting. The closed nesting model treats nested transactions as integral parts of their outermost transactions, i.e., nested transactions have to be executed atomically with their outer-most transactions. In closed nesting, differently from flat nesting, the abort of a nested transaction does not trigger the rollback of its outermost transaction, but only the restart of the aborted nested transaction. This can be beneficial to reduce the abort cost especially in the presence of long-running transactions.

Open nesting. In the open nesting model a nested transaction commits independently of its outer transaction. In other words, a nested transaction is immediately committed, even though later its outer transaction have to be aborted. As such nested transactions do not execute atomically with their outer transactions.

By sacrificing atomicity, the open nesting model provides the opportunity to increase concurrency. However, this comes at the cost of jeopardizing the isolation semantics of the transaction abstraction and typically requires that programmers take extra care to ensure application's correctness.

Sequential vs Parallel Nesting

Another fundamental aspect of transaction nesting is whether a nested transaction executes sequentially in the same thread that generated it (i.e., sequential nesting), or whether it is executed by a different thread and con-

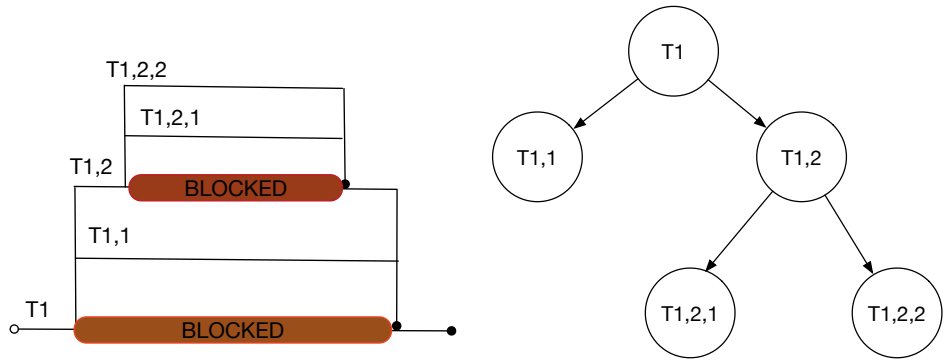
currently with other nested transactions (i.e., parallel nesting).

All the STMs with parallel nesting that we are aware of [68–72] consider a classical fork-join model for the spawning of nested sub-transactions, according to which the spawning/outer (sub-)transaction blocks until all its spawned sub-transaction complete. As such, in the parallel nesting model, the (possibly recursively) nested transactions generated by a top-level transaction can be logically arranged into a tree, which can be used to establish its serialization order. Figure 2.3a illustrates an example execution, in which transaction T_1 forks two nested sub-transactions $T_{1,1}$ and $T_{1,2}$. T_1 is blocked waiting until $T_{1,1}$ and $T_{1,2}$ finish execution. $T_{1,2}$, in its turn, forks two additional sub-transactions, denoted as $T_{1,2,1}$ and $T_{1,2,2}$, and also blocks until both of them complete. The tree shown in Figure 2.3b illustrates the logical dependencies, and corresponding serialization orders, of this set of (sub-)transactions.

Several implementations of the parallel nesting model have been presented in the TM literature, typically considering closed-nesting semantics. Below, three relevant systems are overviewed, namely NePalTM, PNSTM and JVSTM.

NePalTM (Nested PAralleLism for Transactional Memory) [70] exploits OpenMP constructs to allow programmers to expose intra-transaction parallelism opportunity, which is exploited by transparently generating fork-join blocks and trigger the execution of parallel nested sub-transactions. The main limitation of NePalTM is that it only supports one-level of nesting, i.e., all nested sub-transactions have to be spawned from a top-level transaction. This limitation was later overcome by PNSTM (The Parallel Nesting STM) [69], which introduced a novel STM algorithm with support for nesting of arbitrary depths. NePalTM and PNSTM are both locked-based single version implementation. JVSTM (Java-versioned STM), conversely, supports the parallel nesting model while adopting a multi-versioned data model [73].

As already mentioned in Chapter 1, this dissertation studies how to exploit futures to untap intra-transaction parallelism. In this sense, futures can be seen as an alternative to parallel nesting. A detailed discussion on the differences between futures and parallel nesting, as well as on the challenges and opportunities opened by the use of futures in the context of TM will be provided in Chapter 3.



(a) Example execution of fork-join paradigm.

(b) Logical dependencies of the set of sub-transactions.

Figure 2.3: Fork-join Paradigm Gives Rise to a Tree.

2.7 Self-Tuning of TM systems

The efficiency of a TM system is known to be effected by a large number parameters, whose optimal tuning is strongly workload dependent [32, 37]. This has motivated an intense research on how to self-tune different aspects of a TM system in order to ensure robust performance in the presence of heterogeneous workloads.

One example is the work by Felber et al. [54], which proposed a hill-climbing scheme to self-tune several aspects of the management of the locks used to regulate transaction's executed in TinySTM. More in detail, this work aimed to self-tune three parameters of TinySTM: the function used to map memory locations to locks, in order to enhance spatial locality in the access to the internal STM data structures; the size of the lock array, and thus the conflict detection granularity (see Section 2.4.1); the depth of the hierarchical lock array, which allows for trade-off the overhead due to atomic operations and the cost of validation. The settings of these parameters are dependent on the system architecture which have different CPU and cache line size, and the use of self-tuning lift the burden of having to identify the best performing settings from the shoulders of programmers.

Another research line studied how to automatically map threads to physical cores (or hardware threads), possibly in NUMA systems [74, 75]. The self-tuning solutions that approached this problem typically rely on a feed-

back control loop to determine which thread mapping strategy can deliver maximum performance (e.g., throughput). Thread mapping strategy include scattering threads on different sockets of a multi-processor, in contrast to placing them on the same socket or even on sibling hardware threads that share the same underlying physical core, as well as general-purpose, or conventional approaches, e.g., based on round-robin policies.

Other works considered the problem of adapting, and possibly switch at run-time, the underlying implementation of the TM library, e.g., using different STM algorithms [33, 76] or even between STM-based and HTM-based implementations [32, 77]. These systems often exploit machine-learning based approaches (e.g., neural-networks [78] or recommender systems [79] that rely on the availability of a training set, collected off-line in a preliminary phase, by observing the performance of a mix of applications selected to be representative of a wide range of application workloads.

In the context of HTM systems, [80] proposed to identify the optimal retry strategy in presence of different abort types. The retry strategy determines the number of retry before a transaction abort and rollback.

Another problem, which has received significant attention in the literature (and that is closely related to the solution presented in Chapter 6), is how to automatically regulate the degree of parallelism in a TM system. Given the speculative nature of TM, in fact, performance can be severely degraded if an excessively high degree of parallelism is used in the presence of conflict-prone workloads. Several solutions in this arera are based on analytical white box models. The work by Di Sanzo et al. [81], for instance, relies on an analytical model to predict the performance of various STM algorithms. Castro et al. [82] also proposed an analytical model of TM systems but focused on capturing the performance dynamics of hardware based implementations. These modeling tools can be used to estimate the performance of STM applications when using different numbers of top-level threads. Alternatively, other authors have proposed solutions based on black-box approaches. Rughetti et al., for instance, use off-line trained neural-networks to self-tune the concurrency level in STM [9] as well as HTM [7]. In a follow-up work, these pure black-box techniques were combined with an analytical model to reduce the training time [8]. Finally, other works propose solutions based on online learning approaches or customized heuristics. Didona et al. [10] use hill climbing to tune the concurrency level in STM. F2C2-STM (Flux-Based Feedback-Driven Concurrency Control STM) adjusts the parallelism degree according to the target performance

being profiled [83].

Overall, it is important to note that, despite the abundance of works that focused on self-tuning TM systems, none of the existing approaches (that we are aware of) targets the problem of how to self-tune the degree of concurrency in a TM that supports intra-transaction parallelism (e.g., using parallel nesting or futures). This gap in the literature is going to be filled in Chapter 6 of this dissertation.

Chapter 3

The Semantics of Transactional Futures

This chapter focuses on defining desirable atomicity and isolation semantics for TM systems in which futures are used to coordinate the execution of parallel tasks, which we call *transactional futures*, whose accesses to shared data are synchronized via transactions.

In the parallel nesting model, a thread executing in a transactional context (either a top-level transaction or another sub-transaction) can spawn (via language constructs such as *cobegin* and *coend* or *parallelfor*) multiple sub-transactions that are executed in parallel by other threads; the spawning thread remains blocked until the last sub-transaction completes, and only then resumes its execution. Futures, conversely, let programmers activate a parallel task and return a *future contract*, i.e., a promise to deliver the result of the task when the contract is evaluated at a later point in time.

Analogously to parallel nesting, futures allow programmers to express when parallelization is useful: at which point in an otherwise sequential program a parallel task should be started (i.e., when the future is created). Differently from the parallel nesting model, though, the future abstraction does not block the execution of the main thread, also called *continuation* in the context of futures. Further, futures allow a greater flexibility in defining *when* the results of a parallel sub-task are actually required by the application: futures can be evaluated in an asynchronous fashion and in an order that is totally unrelated with the orders in which they were spawned, whereas parallel nesting abides by a fork-join model, which imposes a logical

barrier that enforces the completion of all the sub-tasks.

Using a set of examples of concurrent computations of increasing complexity, we will introduce the spectrum of issues that need to be addressed when defining the semantics of transactional futures. We show that, given the futures' ability to generate parallel computations with complex dependencies, there exist several plausible (i.e., intuitive) alternatives for what concerns the definition of isolation and atomicity semantics between a future and its continuation.

Based on these considerations, we characterize four different semantics over two dimensions: the degree of atomicity between futures and continuations, and their admitted serialization orders. The alternative semantics we propose explore different trade-offs between ease of use (simplicity of reasoning on the equivalent sequential histories), and efficiency (ability to avoid aborts or stalls by enforcing a different type of constraints on the serialization order of transactional futures).

We formalize the semantics utilizing a graph-based characterization of the logical dependencies that (sub-)transactions develop by accessing shared variables and creating/evaluating futures; the resulting graph is then used to formalize alternative definitions of continuations and impose different constraints on the serialization orders of transactional futures.

Some passages in this chapter have been quoted verbatim from [84–87].

3.1 Semantics of Transactional Futures

As a first step to reason on the integration of the future abstraction in the TM paradigm, we first define the assumed model of execution of transaction and futures. We consider a set $\mathcal{TH} = \{Th_1, \ldots, Th_n\}$ of *threads* which can communicate by reading and writing a set of shared variables V.

In the conventional transactional model, transactions start by issuing a *begin* operation, which can be followed by a sequence of *read* and *write* operations, and are finally completed by either a *commit* or *abort* operation. We say that two operations conflict if they access the same variable and at least one of them is a write operation. In case a read operations observes the value written by a write operation, we say that a read-after-write dependence has been developed by the two transactions that issue the read and write operations (or, simply, by the two operations). In order to integrate futures and transactions, we extend this model in a twofold way.

First, we allow transactions to return values: this is done since the future abstraction supports the execution of tasks that generate results, and we intend to encapsulate the operations executed by a future within a transaction. Note that we do not formulate any assumption on the domain of the values returned by transactions nor on the logic used to determine them, e.g., the transactions' logic may be non-deterministic and be affected by the state of the shared variables.

Second, we allow transactions to issue, besides reads and writes, two additional operations: *submit* and *evaluate*. These two primitives allow, respectively, for submitting and evaluating a transactional future, i.e., a transaction encapsulated in a future that can run in parallel with the thread that submitted it. We assume, for simplicity, that future submission and evaluation can only be done within the context of a transaction, this can be straightforwardly enforced, in practice, by wrapping any submit and evaluate call performed in a non-transactional code block within an otherwise empty transaction.

The *submit* operation takes as input a transaction T, activates a parallel thread in which T will be executed, and returns a *future* object $f \in \mathcal{F}$. The returned future f can be passed as an input parameter to an *evaluate* operation to obtain the return value of T.

We assume that a future can only be submitted or evaluated within the context of a transaction. This can be enforced by wrapping any non-transactional submit and evaluate call within an otherwise empty transaction. We initially assume that a future is evaluated at most once and that *evaluate* blocks until the transaction associated with the future f completes its execution. This assumption will be relaxed in Section 3.1.2 .

As in typical TM environments, we assume that if a transaction aborts due to contention, it is re-executed automatically. This implies that if an *evaluate* primitive associated with transaction T returns, then T has either been committed (possibly after several aborts due to conflicts and subsequent re-executions) or T has aborted due to an explicit decision of the program to abort T (via the *abort* operation).

Transactions activated by threads that do not run in the context of a future are denoted *top-level* transactions. This transaction execution model supports an arbitrary deep nesting of calls to transactional futures in a top-level transaction. Also, a transactional future T_F can be uniquely associated with one top-level transaction T_s within whose context T_F is submitted, and with one or more top-level transaction T_e within whose context T_F is evalu-

ated. Importantly, Note also that transactional futures are not required to be evaluated by the same transaction/thread that submit them, i.e., possibly $T_s \neq T_e$.

3.1.1 A Basic Example

Figure 3.1a illustrates a simple example that allows us to set the ground in our search for plausible semantics of execution of transactional futures. The top-level transaction T first writes value 1 to variable X and then submits a transactional future T_F, which reads and increments X by 1. In parallel with T_F, i.e., before evaluating it, transaction T also reads and increments X by 1. Finally, after evaluating T_F, T reads X and writes its value to variable Y.

Given the simplicity of this scenario, it is intuitive to define both which sets of operations should be executed atomically and which are their admissible serialization orders: the read and write operations of T_F should *all* be serialized either before or after the operations of T that follow the creation of T_F and precede T_F's evaluation. We call this set of operations of T the continuation of T_F, and denote it as $\mathcal{C}(T_F)$.

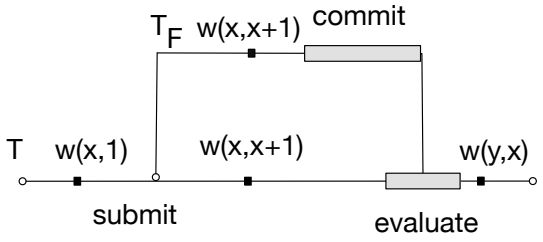
In this example, the serialization orders $T_F \rightarrow \mathcal{C}(T_F)$ and $\mathcal{C}(T_F) \rightarrow T_F$ provide the same outcome, because the operations executed by T_F and $\mathcal{C}(T_F)$ commute. Clearly this may not the case in general. In cases where the serialization order of T_F and $\mathcal{C}(T_F)$ is relevant, it is desirable to allow programmers to specify restrictions on the serialization order of transactional futures. In this sense, to ensure equivalence with a sequential version of the program (not using futures), a simple solution is to impose the serialization of the operations of T_F before the ones of $\mathcal{C}(T_F)$.

These considerations lead us to consider two different semantics regarding the plausible serialization order of transactional futures and their continuations

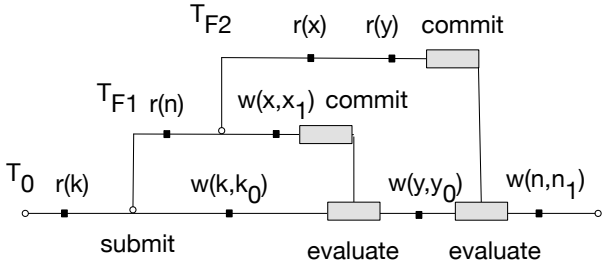
- *Weakly Ordered Transactional Futures (WO)*: A future and its continuation should appear as executed atomically, i.e. the future should be serialized before or after its continuation.

- *Strongly Ordered Transactional Futures (SO)*: A future and its continuation should appear as executed atomically with the future serialized before its continuation.

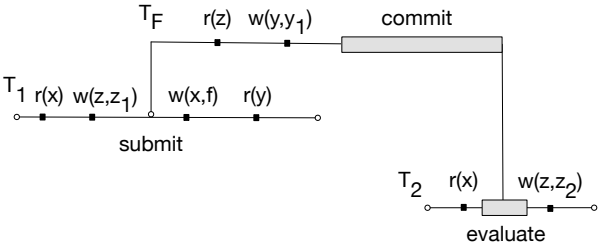
The SO semantic ensures that a transactional future yields the same



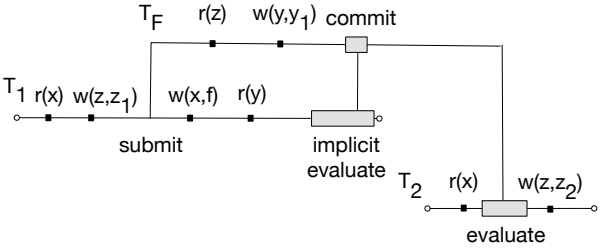
(a) A Simple Example of Transactional Futures (TF).



(b) Escaping TF that is evaluated within the same top-level transaction.



(c) Escaping TF across different top-level transactions with Globally Atomic Continuation semantic.



(d) Same history as in Figure 3.1c but with Locally Atomic Continuation semantic.

Figure 3.1: Example executions with Transactional Futures.

result as if it executed in a sequential version of the program (not using futures). WO semantics, conversely, require programmers to determine whether application's correctness is preserved independently of the order in which transactional futures and their continuations are serialized.

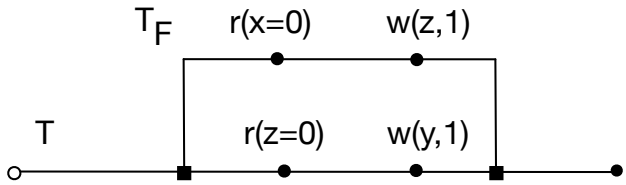


Figure 3.2: The continuation aborts with SO semantics and commits with WO semantics.

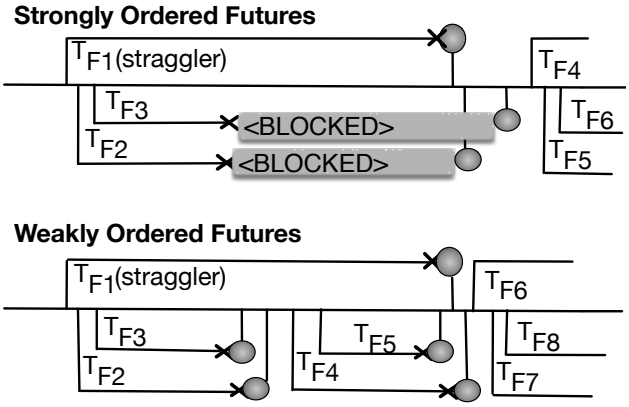


Figure 3.3: The SO semantic exposes to the risk of stragglers, which can be avoided with WO.

On the other hand, the ability of WO to establish different serialization points for a future brings two benefits:

• *Reduction of abort rate*: as exemplified in Figure 3.2, the continuation of T_F can be spared from aborting in case it misses to observe the updates produced by T_F (whereas the continuation would have to be aborted with SO semantics in such a history).

• *Stragglers avoidance*: with SO semantics, a transactional future, T_F^i, can only be committed if any previously submitted future, say T_F^j with $j < i$

(where the superscript denotes the spawning order of the transactional future), has first completed its own execution. As such, even a single relatively slow future can become a straggler for the whole set of futures concurrently submitted by the same top-level transaction. This phenomenon is exemplified in Figure 3.3, which illustrates a scenario in which a top-level transaction, logically composed by a total of 8 (commutative) sub-tasks, is parallelized using up to 3 concurrent futures, i.e., a new future is activated only whenever the continuation detects that a previously submitted future has completed its execution. The diagram clearly illustrates that, thanks to the use of WO semantics, the heterogeneity of the execution speed of transactional futures does not expose the system to the risk of stragglers.

Further, the choice of WO versus SO has implications on the definition of the upper bound of the execution interval of the *commit* operation of a future and of its spawning transaction.

With SO semantics the serialization order of a future is defined *prior* to the future's activation. As such, whenever a SO future requests to commit, it is immediately possible to determine the outcome of the future (i.e., if this can be serialized upon submission) and return from the commit call. Conversely, the SO semantic demands that any future spawned by a transaction T is serialized before its continuation, i.e., within T. It follows that T's commit request has to be necessarily blocked until all the futures spawned by T have committed.

With WO semantics, the opposite is true: a transaction T that spawns a future T_F (and does not evaluate it) can commit without waiting for T_F, as T_F can be serialized upon evaluation, i.e., after T. However, whenever a future is serialized upon evaluation, the return call of its *commit* operation must follow the call of its *evaluate* operation (else, its serialization point would be undefined). Thus, the T_F's commit request may be blocked for an arbitrarily long time, i.e., until T_F is evaluated.

The latter case is illustrated in Figure 3.1a, which depicts the execution interval of the *commit* and *evaluate* operations of T_F (for simplicity all the other operations are assumed instantaneous). In this example, T_F requests to commit in real time before it is evaluated and assumes that the TM opted for serializing T_F upon evaluation. As such, the commit request of T_F has to be blocked until T_F is evaluated by T.

3.1.2 Non-blocking and repeated evaluations

So far we have assumed that *evaluate* calls are blocking, i.e., they only return if the future being evaluated has finalized its execution. Supporting a non-blocking variant does not raise significant issues in any of the proposed semantics. In fact, as any attempt to evaluate a future that is still executing has no impact on its possible serialization orders (since the call returns without externalizing the future's results).

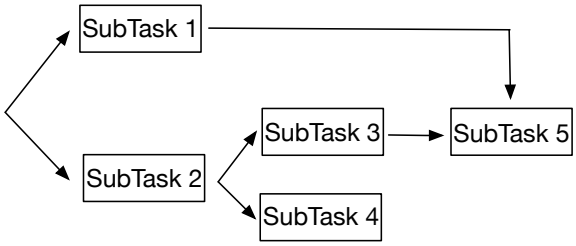
Let us now discuss the semantics of evaluating a future multiple times. With strongly ordered semantics, evaluate calls do not affect the serialization order of futures. Thus, supporting multiple evaluations is natural: all of them behave in the same way. With weakly ordered semantics, though, evaluation calls can affect the serialization order of futures, so it is relevant to specify the semantics of multiple future evaluations. The semantics we propose is based on the common assumption that a transaction is only committed (and, hence, serialized) once. Accordingly, only the first (blocking) *evaluate* called after the completion of the future determines the serialization point of transactional futures in weakly ordered semantics. Further evaluations just return the results the future produced during its first evaluation.

3.1.3 Beyond parallel nesting

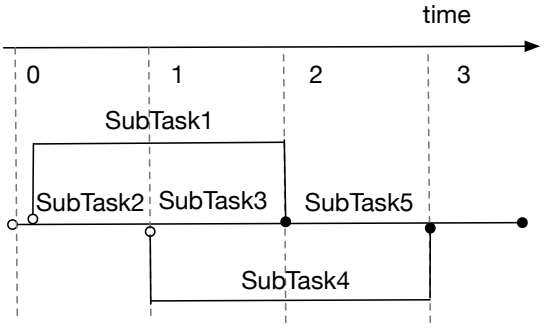
The simple example considered above could also have been implemented using parallel nesting [68] based on the classic fork-join model. However, futures support a broader class of concurrent computations than parallel nesting. Unlike parallel nesting, in fact, futures do not force blocking the "main" thread until *all* its nested transactions complete their execution, but rather allow for the arbitrary interleaving of submissions and evaluations of different futures.

The following is a first example that demonstrates how the future abstraction can support more efficient execution schedules when compared to parallel nesting.

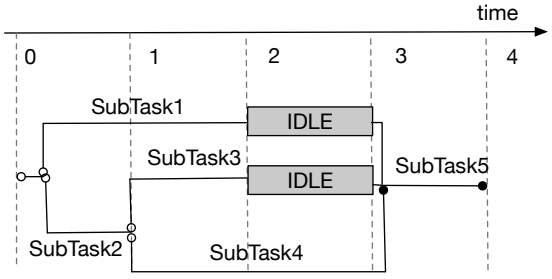
Consider a long-running job that can be decomposed into 5 sub-tasks, whose logical dependencies are shown in the DAG (Directed Acyclic Graph) in Figure 3.4a, e.g., sub-tasks ST5 takes as input the results produced by sub-tasks ST1 and ST3, while ST3 and ST4 depend on ST2. Assume that the execution time of all sub-tasks is one time unit except for ST1 and ST4,



(a) Dependency graph of 5 sub-transactions.



(b) Execution of the sub-transactions in Figure 3.4a using futures.



(c) Execution of the sub-transactions in Figure 3.4a using parallel nesting.

Figure 3.4: Demonstrating benefits of futures over parallel nesting.

whose execution lasts 2 time units.

Figure 3.4b and Figure 3.4c illustrate the most efficient schedules of the sub-tasks in Figure 3.4a that can be generated using futures and parallel nesting, respectively. As already mentioned, with parallel nesting the spawning thread is forced to block until *all* the sub-task it spawned complete execution. Accordingly, new sub-task can only be spawned from within other sub-tasks. As a result, one can not spawn a sub-task that depends on two (or more) of the active sub-tasks, until all of their concurrent sub-tasks complete.

This restricts the set of possible parallel schedules that can be produced using parallel nesting, which can lead to inefficient mapping of the sub-tasks dependency graph and cause longer makespans (i.e., 4 vs 3 time units, in this example) when compared to futures: since with futures the spawning thread can execute concurrently with its nested sub-tasks, it can wait selectively for some of the futures it spawned, and therefore avoid unnecessary stalls.

Regarding atomicity between futures and continuations, we argue that in this scenario the natural semantics is to enforce the atomicity of the operations of the future ST1 with respect to both ST2 and ST3, which represent its continuation, i.e., ST1 should be serialized before ST2 and ST3, or after both of them. Analogously, the future ST4 should either observe the effects of both ST3 and ST5 (since they represent its continuation) or of none of them. Note that the continuations of ST1 and ST4 are partially overlapping (they share ST3), yet distinct.

In the following we focus on another type of schedules that are allowed by the assumed execution model of transactional futures and that cannot be supported with parallel nesting, which we call *escaping transactional futures*, i.e., transactional futures that are not evaluated by the same transaction in which they are submitted.

We show an example of escaping futures in Figure 3.1b, in which T_{F2} is activated by the T_{F1}. The latter a write to X, but commits without evaluating T_{F2}, whose reference is communicated to T_0 via T_{F1}'s return value. Next, T_0 issues a write on Y and evaluates T_{F2}. This example highlights that the definition of continuation for escaping transactional futures is very subtle. We argue that, in this case, the natural continuation of T_{F2} (i.e., the sequence of causally-related operations that leads from the start of T_{F2}'s continuation to its evaluation) is composed by the write on X by T_{F1} *and* by the write on Y by T_0, i.e., T_{F2}'s continuation spans two (sub-)transactions (associated with the same top-level transaction). Hence, the reads issued by

T_{F2} should observe either both writes on X and Y or none of them.

Figure 3.1c depicts another interesting programming pattern in which escaping transactional futures are used as a complementary communication means (in addition to shared variables) by two distinct top-level transactions. In this case, the top-level transaction T_1 submits a future T_F. In T_F's continuation, T_1 writes a reference of the future returned by $submit(T_F)$ to variable X, reads Y and commits. If WO semantic is specified for T_F, it is possible to serialize T_F after the write and read by T_1. This allows the top-level transaction T_1 to commit without having to block waiting for T_F to commit first[1], making the reference to T_F available to other top-level transactions possibly executing on different threads, e.g., T_2 in Figure 3.1c.

Following the same rationale used when analyzing the example of Figure 3.1b, one may argue that by communicating the reference of T_F via X, a logical causality has been established between the write operation to X and read operation to Y by T_1 and the operations issued by T_2 before evaluating T_F. Despite spanning two top-level transactions, these operations represent a chain of causally related events that led to the evaluation of T_F. If they were, in the light of this reasoning, considered as continuation of T_F, then they would have to appear as an atomic block to T_F. We term this atomicity model *Globally Atomic Continuation* (GAC).

However, the above example could be easily generalized to include in the continuation of T_F, after T_1 and before T_2, an arbitrarily long chain of sequential transactions propagating the reference to T_F to each other. As already discussed in Section 3.1.1, with WO semantics, this would force a TM that opts for serializing T_F upon evaluation to stretch the execution interval of its commit operation for an arbitrarily long time, i.e., until T_F is evaluated. This can have implications both on the efficiency of TM implementations, which will have to maintain resources (e.g., locks held by the future) for prolonged periods of time, and on the likelihood for the T_F to be aborted (since by stretching its execution interval, T_F becomes more likely to conflict with concurrent transactions).

This led us to consider an alternative atomicity semantic, called *Locally Atomic Continuation* (LAC), which limits the boundaries of a continuation to its spawning top-level transaction. With this semantic any top-level trans-

[1]This would be necessary if we assumed SO semantic for T_F. In this case, it cannot be otherwise guaranteed that T_1 observes all the writes possibly issued by T_F unless T_F completes its execution.

action T is requested, during its commit phase[2], to *implicitly* evaluate any escaping futures F it spawned, either directly or indirectly (i.e., T triggered the spawning of a chain of futures that led eventually to the submission of F). We call this evaluation "implicit", since it is not requested explicitly by programmers but is rather imposed by the considered atomicity semantic.

Note that the LAC semantic ensures that a future is necessarily serialized within its spawning top-level transaction. In fact, with LAC, a future that escapes from its top-level transaction T is serialized either upon submission or (if WO semantic is assumed) upon its "implicit" evaluation, i.e., as the last (sub-)transaction of T right before T's commit.

Figure 3.1d illustrates the LAC semantics for the same history considered in Figure 3.1c: an implicit evaluation of T_F is added as the last operation of its spawning top-level transaction T_1. As a consequence, T_F can commit much earlier than if GAC semantics were considered. Note, though, that this come at a cost for T_1, which is now forced to wait for T_F's completion before being able to commit. Figure 3.1d also shows an example of repeated evaluations of a future, namely T_F, which is first implicitly evaluated by T_1 and then (explicitly) evaluated by T_2. As discussed in Section 3.1.2, the second evaluation is required to return the same value as in the first evaluation.

Finally, let us discuss the relations between the proposed semantics, which were defined over two dimensions: the atomicity between transactional futures and continuations (GAC versus LAC), and their admitted serialization orders (WO versus SO). We note that the choice of SO semantics, which demand futures to be serialized at submission time, renders the distinction between globally and locally atomic continuations irrelevant: establishing which set of operations to include in a future's continuation is only relevant if futures can be serialized at their evaluation, i.e., after their continuation, as allowed by the WO semantics; with SO semantics, in fact, a future's serialization point is known *a priori* and is not affected by the set of the operations included in its continuation.

3.1.4 Formalizing the Semantics of Transactional Futures

The first step to formalize the concept of atomicity between transactional futures is to introduce a framework to establish the set of feasible serializa-

[2]If multiple transactional futures escape from the same top level transaction, the order in which they are implicitly evaluated is irrelevant.

tion orders among transactional futures and continuations. To this end we introduce a graph-based characterization, called Future Serialization Graph (FSG), that captures the possible serialization orders among transactional futures and continuations. Let $\mathcal{H}(\mathcal{T}, \mathcal{S})$ be a history defined over: i) a set of transactions $\mathcal{T} = \mathcal{T}_{top} \cup \mathcal{T}_{fut}$, where $\mathcal{T}_{top}$ denotes the set of top-level transactions and $\mathcal{T}_{fut}$ denotes the set of transactional futures; ii) a partial order $\mathcal{S}$ over the operations issued within each transactions in $\mathcal{T}$, extended to include real-time order relations between transactions.

FSG($\mathcal{H}$) is defined as a directed graph having the following set of vertexes:

- A vertex V_T^{begin} for each transaction $T \in \mathcal{T}$, which is associated with all the operations executed by T since its begin until (and including) the first occurrence of the first of the following operations {*submit*, *evaluate*, *abort*, *commit*}.
- For each transactional future $T \in \mathcal{T}_{fut}$, two additional vertexes are defined:
  1. $V_T^{C-begin}$, which is associated with the sequence of read, and write operations executed by the thread that submitted the future T (i.e., the initial part of T's continuation) since T's submission (excluded) until the first occurrence of a {*submit*, *evaluate*, *abort*, *commit*} operation. This last operation is associated with $V_T^{C-begin}$, except in case it is an *evaluate* operation, for which a dedicated vertexes (V_T^{eval}, defined next) is added to the graph.
  2. V_T^{eval}, which is associated with the sequence of operations issued by some thread that starts with the (possibly implicit) evaluation of Future T (included) and ends with the first occurrence of a {*submit*, *evaluate*, *abort*, *commit*} operation. Also in this case, this last operation is associated with V_T^{eval}, except in case it is an *evaluate*(T') operation, in which case a new $V_{T'}^{eval}$ vertex is added to the FSG.

The following edges are defined for FSG($\mathcal{H}$):

1. An edge $V1 \rightarrow V2$ for each pair $V1$, $V2$ of vertexes in FSG such that $V1$ and $V2$ are executed by the same thread $Th_i \in \mathcal{TH}$ and Th_i executes $V1$ before $V2$, which captures the sequential order of execution of the operations issued by a single thread.

2. An edge $V_T^{spawn} \rightarrow V_T^B$ for each transactional future $T \in \mathcal{T}_{fut}$, where we have denoted with V_T^{spawn} the vertex associated with the operation *submit*(T), i.e., transactional futures cannot be serialized before their submission.

3. An edge $V_T^{end} \rightarrow V_T^{eval}$ for each transactional future $T \in \mathcal{T}_{fut}$, where we have denoted with V_T^{end} the vertex associated with the operation *commit*(T), i.e., transactional futures cannot be serialized after their evaluation.

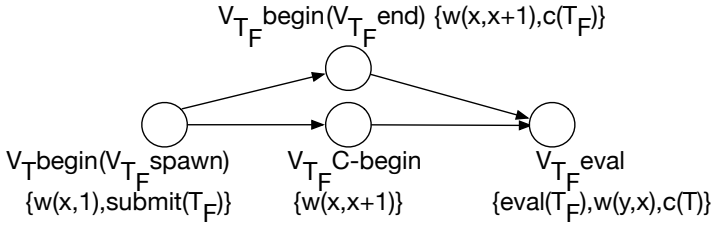
For the sake of clarity, Figure 3.5a shows the FSG of the example reported in Figure 3.1a. V_T^{begin}, which coincides with $V_{T_F}^{spawn}$, is associated with the operations *write*(x,1), *submit*(T_F); $V_{T_F}^{begin}$, which coincides with $V_{T_F}^{end}$, is associated with the operations *write*(x,x+1), *commit*(T_F); $V_{T_F}^{C-begin}$ is associated with the operation *write*(x,x+1); $V_{T_F}^{eval}$ is associated with the operations *eval*(T_F), *write*(y,x), *commit*(T). Figure 3.5b illustrates a slightly more complex FSG associated with the history in Figure 3.1c with GAC semantic. Figure 3.6 shows also the FSG for the history of Figure 3.1d with LAC semantic.

Weakly and Strongly Ordered Futures. We now extend the FSG with additional edges aimed at imposing restrictions on the serialization order of every future with respect to its continuation, depending on the considered (strongly versus weakly) ordered semantics.

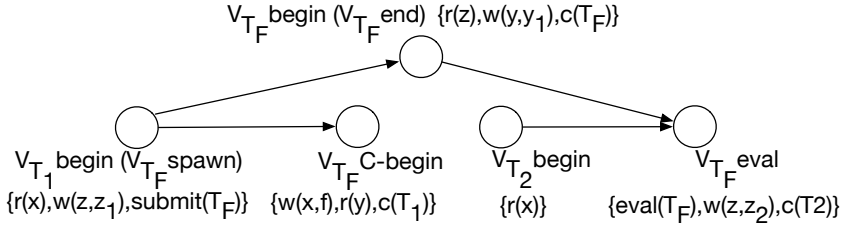
Recall that the SO semantic requires a future to appear as atomically executed with its spawning transaction and logically serialized at the time of its submission. This semantic can be easily encoded in the FSG by adding an edge $V_T^{end} \rightarrow V_T^{C-begin}$ for each SO transactional future $T \in \mathcal{T}_{fut}$, where we have again noted with V_T^{end} the vertex associated with the *commit* operation of T. Intuitively, this edge ensures that SO transactional futures are serialized before their respective continuations. This is illustrated in Figure 3.5c, where the edge from $V_{T_F}^{begin}$ (which coincides with $V_{T_F}^{end}$) to $V_{T_F}^{C-begin}$ is used to serialize T_F before its continuation.

The WO semantic, on the other hand, dictates that the serialization point of a transactional future is either upon its submission or its evaluation.

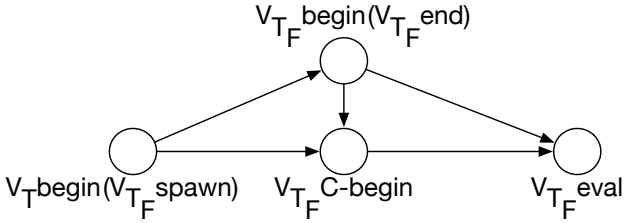
To enforce this constraint, we introduce a special type of edge, called bipath [18]. The notion of bipath was introduced by Papadimitrou in his seminal paper on the complexity of (view) serializability. A bipath is defined by a pair of edges $(V_i \rightarrow V_j)$,$(V_k \rightarrow V_l)$ and the inclusion of a bipath in a



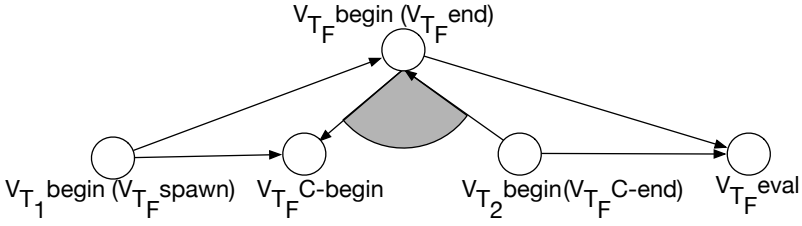
(a) FSG of the History in Figure 3.1a (no ordering semantics).



(b) FSG of the history in Figure 3.1c (no ordering semantics).



(c) Extending the FSG of the History in Figure 3.1a with an edge from $V_{T_F}^{end}$ to $V_{T_F}^{C-begin}$ imposes SO semantic to T_F.



(d) Extending the FSG of the History in Figure 3.1c with a bipath to impose WO semantics to T_F.

Figure 3.5: Example of FSG-based representations.

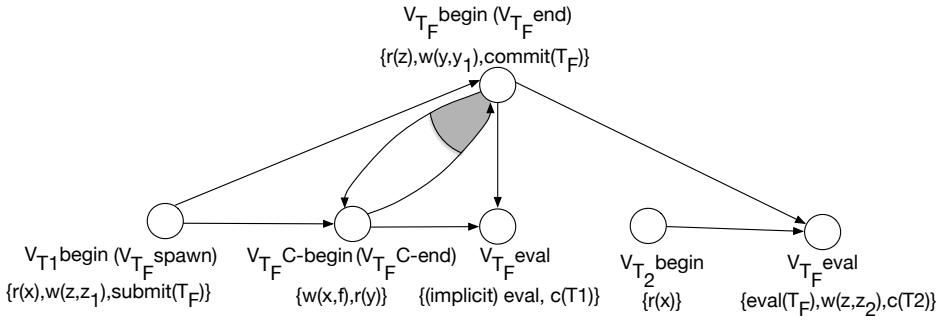


Figure 3.6: FSG of the History in Figure 3.1d with WO semantic.

graph serves to express that either the first or the second edge of the bipath holds. More formally, the inclusion of a bipath in a directed graph, such as the FSG, turns the graph into a *polygraph*. A polygraph encodes a family of directed graphs, where each directed graph is obtained by including, for each bipath in the original polygraph, either one of its edges. As such, polygraphs allow for representing in a compact way a large number of directed graphs. More precisely, a polygraph with n bipaths encodes 2^n different directed graphs [18].

We capture the two alternative serialization orders admitted for a WO transactional future by introducing for each transactional future T_F a bipath:

$$(V_{T_F{}^{C-end}} \rightarrow V_{T_F{}^{begin}}), (V_{T_F{}^{end}} \rightarrow V_{T_F{}^{C-begin}})$$

where $V_{T_F{}^{C-end}}$ represents, intuitively, the final vertex of the continuation of T_F or, more formally, the vertex in the FSG associated with the operation that immediately precedes the evaluation of T_F in the (totally ordered) history of operations of the thread that evaluates T_F. With the above definition, the first edge of the bipath orders the continuation before the future (i.e., serialization upon evaluation) and the second edge orders the future before the continuation (i.e., serialization upon submission).

Figure 3.5d illustrates how the FSG for the history in Figure 3.1c is extended with a bipath that connects: (i) the end vertex of T_F's continuation (i.e., $V_{T_2}^{begin}$) to the beginning vertex of T_F (i.e., $V_{T_F}^{begin}$) and (ii) the end vertex of T_F (which coincides with $V_{T_F}^{begin}$) to the beginning vertex of T_F's continuation (i.e., $V_{T_F}^{C-begin}$).

Inclusion of operations in transactions. Before finalizing the formalization of the proposed semantics for transactional futures, we need to introduce the notion of inclusion of operations into a transaction. The intuition is to include in a transaction T not only the operations that T directly executes, but also the operations executed by transactional futures that are serialized within the context of T possibly indirectly, i.e., via a chain of intermediate futures that are spawned/evaluated by T.

An operation *op* is included in a transaction T if there exists a path in the FSG from the vertex associated with the begin operation of T to the vertex associated with the commit/abort of T that passes via the vertex associated with *op*. Based on this definition, we include in a transaction T all the operations issued by T and by any non-escaping future submitted by T and, recursively, by T's non-escaping futures. On the other hand, with WO semantics, a transactional future that escapes from its spawning transaction and is evaluated in a different transaction is given the flexibility to be either included in its spawning or in its evaluating transaction.

Extending the FSG with conflict relations. We complete the formalization of the proposed semantics for transactional futures by introducing an additional extension of the FSG that aims at capturing the conflict relations between pairs of transactions in $\mathcal{T}$. More precisely, whenever a TM executes an operation *op* issued by a transaction T the following edges are added to the FSG to track conflict relations at the level of both individual transactions (e.g., two futures included in the same top-level transaction) as well as the top-level transactions in which the conflicting transactions are included:

- *Atomicity between individual transactions.* If *op* conflicts with an operation op' *executed* by a different transaction T', an edge is added from all the vertexes associated with T to all the vertexes associated with T' or in the opposite direction, depending on whether the TM orders *op* before or after op'.
- *Atomicity between different top level transactions.* If T is included in a top level transaction T_{top} and *op* conflicts with an operation op', where op' is included in a different top-level transaction T'_{top}, an edge is added from all the vertexes associated with T_{top} to all the vertexes associated with operations included in T'_{top} or in the opposite direction, depending on whether the TM orders *op* before or after op'.

Next, as in classical serializability theory [18], we impose the constraint

that an operation can be accepted iff its execution does not generate cycles in the FSG (else the transaction that issued the operation has to be aborted). Intuitively, the specification of this constraint guarantees the equivalence of the history $\mathcal{H}(\mathcal{T}, \mathcal{S})$ produced by a TM to a sequential history that respects not only the partial order $\mathcal{S}$ of the operations in $\mathcal{H}$ but also of every additional constraint imposed by the semantics of weakly versus strongly ordered transactional futures and of globally versus locally atomic continuations.

Note that the proposed formalization requires that the absence of cycles in the FSG is ensured before allowing any operation issued by a transaction to return, and not only when a transaction is committed. In this sense, the proposed semantics for transactional futures are similar in spirit to existing safety criteria for TM systems that do not support futures, such as opacity [39] or virtual world consistency [88], which also aim at preventing active transactions, even those that eventually abort, from observing arbitrary snapshots not producible by a sequential execution of some subset of the committed transaction.

Finally, it should be noted that, if WO semantics are used, then the FSG is not a plain directed graph, but a polygraph that encodes a family of directed graphs. In this case, a history can be accepted iff there exists at least one directed graph encoded by the polygraph that contains no cycles. This definition of acyclicity of polygraphs, which coincides with the one used by Papadimitrou [18], intuitively captures the fact that for at least one of the set of viable serialization orders of the WO futures in the history (each encoded by a different directed graph), it is possible to prove the existence of an equivalent sequential history.

3.1.5 Trade-offs between atomicity and isolation

Let us now analyze the relations and trade-offs between the various types of semantics we proposed, which, we recall, are defined over two dimensions: the degree of atomicity between transactional futures and continuations, and their admitted serialization orders.

We start by observing that these two dimensions are not completely independent. In fact, the choice of strongly ordered semantics, by imposing serialization of futures to take place solely at submission time, renders the distinction between globally and locally atomic continuation irrelevant: establishing which set of operations should be considered as included by a future's continuation is only relevant if futures can be serialized at their

evaluation, i.e., after their continuation, as allowed by the weakly ordered semantics.

Let us now focus on analyzing the trade-offs associated with the choice of strongly and weakly ordered semantics. As already mentioned, strongly ordered semantics ensure that futures can be seamlessly employed to parallelize arbitrary sequential code. But if strongly ordered semantics have benefits for what concerns ease of use from the programmer's perspective, their inflexibility makes them also inherently prone to incur run-time overheads that can instead be avoided embracing weakly ordered semantics. In particular, the possibility, enabled by weakly ordered semantics, to establish different serialization points for a given transactional future can lead to two benefits in terms of efficiency, namely, reduction of transaction's abort rate and stragglers avoidance, as discussed in Section 3.1.1.

3.2 Conclusion

This chapter addressed the problem of integrating futures and transactional memory, proposing a set of semantics for transactional futures that explore different trade-offs between ease of use and efficiency.

As a first step to formalize the atomicity between transactional futures, we have introduced the Future Serialization Graph (FSG). Via the FSG, we specified how the boundaries of a future's continuation should be defined by proposing the notion of globally and locally atomic continuations.

We also established the set of feasible serialization orders among transactional futures and their continuations by formalizing two alternative semantics that define plausible serialization order of transactional futures and their continuations: (1) Strongly Ordered Transactional Futures (SO), which requires that a transactional future is serialized before its continuation (2) Weakly Ordered Transactional Futures (WO), which requires that a transactional future is serialized either before or after its continuation.

In the following two chapters, we present how to implement the proposed transactional future semantics, both SO and WO, in state of the art TM systems and what efficiency such implementations can achieve in practice.

Chapter 4

An Implementation of the Strongly Ordered Transactional Future Abstraction

In this chapter we present the design of a software-based TM, called Strongly Ordered Java Transactional Futures (SO-JTF), that implements the proposed SO (Strongly Ordered) transactional future abstraction for the Java run-time. The SO-JTF system transparently wraps futures and continuations into atomic blocks, which run as parallel sub-transactions and are guaranteed to be serialized as if they were instantaneously executed at the moment in which they were spawned.

We evaluated SO-JTF by parallelizing, using transactional futures, two popular benchmarks for transactional systems (i.e., both TM and database systems) and tested it using a 48-core machine. The results of our experimental study show that, by exploiting intra-transaction parallelism, SO-JTF can not only significantly reduce the execution time of long running transactions, but also strongly benefit throughput in contention-prone workloads, by reducing the likelihood of conflicts among transactions and the performance penalty due to aborts.

4.1 Overview of The SO-JTF system

Analogously to other Java-based STM [35, 73], SO-JTF requires programmers to explicitly identify the objects whose accesses need to be tracked by the transactional system. This is achieved by storing references to objects or primitive data types within a data container, called *VBox* (versioned box), that exposes methods to retrieve (*get*) and update (*put*) the current value of the object. These methods are intercepted by the SO-JTF runtime, which uses them to track and regulate concurrent data accesses from within the transactional contexts.

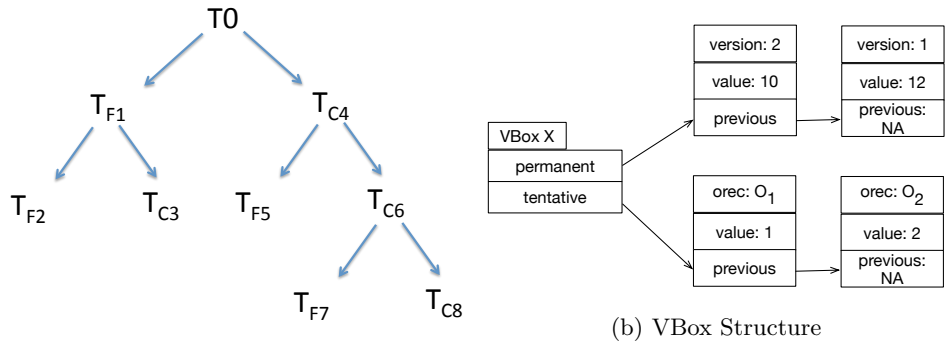
Like classic TM systems, SO-JTF exposes methods to begin, commit and abort transactions. In addition to conventional TM systems, though, it provides methods also to request the execution of a method (via the standard *Callable* interface in Java) as a transactional future, and to evaluate transactional futures' results. When a transactional future is submitted, the SO-JTF runtime schedules its execution on an internal thread pool and transparently wraps the execution of the future's method in the context of a new (sub-)transaction. Further, SO-JTF also starts another child transactional context to run the continuation. In this way, SO-JTF can discard all the effects produced by the continuation and still preserve the effects of the parent transaction before the invocation of the transactional future (i.e., support partial rollback).

SO-JTF allows for evaluating the future's result at any time, without requiring the evaluation to occur from within a transactional context. The evaluation semantic of a transactional future is identical to that of a plain future, blocking the calling thread until the future's execution has completed. A reference to the future can be propagated among different threads (e.g., via writes to shared memory), which allows to use transactional futures as a channel to support inter-thread communication. This mechanism does not interfere with Java's garbage collection mechanism, as a transactional future and its result can be safely garbage collected as soon as no other object stores its reference.

4.1.1 Concurrency control

We start by discussing how SO-JTF treats top-level transactions, then we discuss how transactional futures are handled.

Top-level transactions. Every top-level transaction in SO-JTF has a



(a) Transaction Tree

(b) VBox Structure

Figure 4.1: Data Structure under JTF System.

version number which is assigned when the transaction is created, and establishes the data snapshot visible to that transaction during execution. This number is fetched from a global counter that represents the version number of the latest read-write transaction that successfully committed. Child transactions also receive a version number, which they inherit from the parent transaction.

SO-JTF stores in the VBoxes the set of committed (or permanent) data versions that may be required to process reads from concurrent transactions. The committed versions are stored in a sorted list, which we call *permanent list*, in descendent order of recency and are tagged with a version number that defines the serialization order of the top-level transaction that committed this version (see Figure 4.1b).

The write operation of a top-level transaction is implemented by simply storing the value in a private write-set. Reads are managed by first doing a lookup in the transaction's write-set, and, if the requested data item is not found there, the most recent version created by a transaction that committed before the transaction started is returned.

SO-JTF integrates a lock-free commit algorithm, first introduced in JVSTM [35], which uses a helping mechanism to implement the following two steps in a non-blocking, yet atomic, fashion: increasing the global counter and writing-back the values from the transaction's write-set to the corresponding VBoxes.

Transactional Futures: submission. As already mentioned, SO-JTF

transparently wraps the execution of a transactional future and its continuation into two sub-transactions. These sub-transactions inherit the *version* number from their parent, which ensures that they will be serialized in the same order as their parent with respect to other top-level transactions.

SO-JTF supports partial roll-back of a transactional tree, e.g., if a continuation misses the write generated by its corresponding future[1], only the sub-tree rooted on the continuation sub-transaction is aborted. To this end, the SO-JTF run-time checkpoints the control state of the thread submitting the future, by means of a *first-class-continuation* (FCC) [89]. FCCs should not be confused with the continuations of (transactional) futures; FCCs are a lightweight check-pointing mechanism that allows to reify, save and restore the program control state (e.g., its stack and the JVM registers' value). In SO-JTF, we use the FCC support provided by the OpenJDK Hotspot VM [89], which introduces very limited overhead and, unlike other approaches [90], does not require any byte-code rewriting.

Transactional Futures: read and write operations. Unlike top-level transactions, sub-transactions do not maintain a private write-set. When a sub-transaction writes, it acquires a lock (valid for the entire transactional tree) and inserts its version in a second list, called *tentative list*, in the corresponding VBox (see Figure 4.1b). Unlike the versions in the permanent list, each tentative version is associated with the *orec* (ownership record [91]) of the (sub-)transaction that created it. The *orec* maintains a pointer to the (sub-)transaction (the owner), the version of the write (*txTreeVer)* and the owner status. Each sub-transaction creates the *orec* when the write occurs and propagates it to its parent when the sub-transaction commits.

In order to determine which tentative data item versions are visible to the sub-transactions of a transactional tree, sub-transactions maintain two additional meta-data: *nClock*, a counter initialized with 0 and incremented whenever a direct child transaction commits; *ancVer*, a map computed when the sub-transaction starts, which includes the parent's *ancVer* extended with the parent's current *nClock* value.

A sub-transaction T can only observe the tentative version of a transaction T' if, at the time in which T started: i) T' has already committed, and ii) the commit of T' has been already propagated to an ancestor of T. This

[1]We say that a continuation, T_C, misses the write of its corresponding future, T_F, if T_C issues a read on a data item x, for which a new version, x_F, is produced by T_F, and T_C does not return x_F.

is verifiable by checking in the *ancVer* of T whether the owner of the *orec* of a data version is an ancestor of T ($orec.owner \in T.ancVer$), and verifying whether such ancestor had already witnessed the commit of T' at the time in which T had started ($orec.txTreeVer \leq T.ancVer(orec.owner)$).

This mechanism is illustrated in Figure 4.2, which depicts the metadata maintained at a given point of execution of the transactional tree originally considered in Figure 4.1a. $T0$'s *nClock* reflects the commit of its left sub-tree, having as root T_{F1}. By T_{C4}'s *ancVer*, we get that when T_{C4} started, T_{F1} had already propagated its commit to $T0$. T_{C4} and its descendants can hence observe the tentative writes of any of the sub-transactions of the left sub-tree of $T0$. T_{C6}, however, cannot see the tentative versions of T_{F5}, although T_{F5} has already committed. This is because at the moment in which T_{C6} started, T_{F5} had not yet committed (as the T_{C4}'s entry in the *ancVer* of T_{C6} is 0).

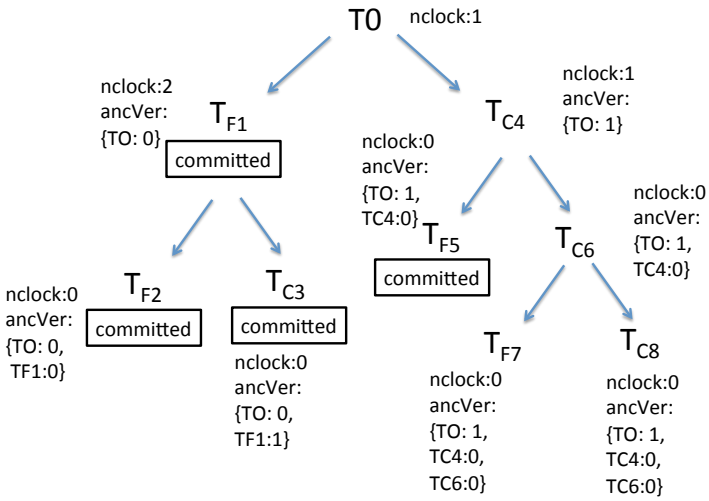


Figure 4.2: Metadata reflecting the visible snapshots in a transactional tree.

Transactional Futures: commit phase. Whenever a sub-transaction (either a transactional future or a continuation) finishes its execution it must then check for conflicts that may have broken the sequential semantics of the top-level transaction's code. However, there is a sequential dependence between transactional futures and continuations. In practice, this implies

that when a sub-transaction running a transactional future or a continuation finishes execution, and before it validates, it must wait that all other sub-transactions that precedes it according to the strongly ordered semantics have validated and committed.

When a child transaction T reaches its turn to commit, it must then check if there is an intersection between its reads and the writes of some other sub-transactions (in the same transaction tree) that have committed while T was executing. In that case, it means that a conflict that broke the sequential semantic occurred, and T must be re-executed.

As already mentioned, the commit procedure of a sub-transaction (either running a transactional future or a continuation) ensures that the writes it performed are propagated to the parent transaction, which becomes the new owner of these versions. From that point on, those writes are made visible to new child transactions the parent might spawn. This allows transactions that re-execute, due to a conflict, to read the writes they missed on their previous execution.

Once all transactions in the transactional tree have committed, the control is passed to the top-level transaction. At that point, the top-level transaction will validate its execution against other top-level transactions in the system and commit.

4.2 SO-JTF's concurrency control in detail

This section presents the pseudo-code of SO-JTF's concurrency control algorithm. We describe the behaviour of transactional futures and continuations, omitting the pseudo-code of top-level transactions.

4.2.1 Write operations

Algorithm 4.1 reports the pseudo-code for managing the write operation of a sub-transaction. When writing to a *VBox*, the (sub)transaction T fetches the tentative version at the head of the tentative list and reads its ownership record *orec* to tell whether it owns that version or not (line 7). In this positive case, it simply overwrites the previous write.

Otherwise, T checks if the transaction that created that tentative version has already completed execution, in which case T attempts to acquire ownership of the tentative write at the head of the list (line 11) using a compare-and-swap (CAS). If the CAS fails, we know some other transac-

Algorithm 4.1 Write Procedure by Transactional Future or Continuation

```
 1: Write(T,vbox,value):
 2: pointerWrite ← vbox.tentative
 3: orec ← pointerWrite.orec
 4: status ← orec.status
 5: if orec.owner == T then
 6:             ▷ T owns the tentative version
 7:     pointerWrite.value ← value
 8:     return
 9: end if
10: if status ≠ RUNNING then
11:     if pointerWrite.CASorec(orec,T.orec) then
12:         pointerWrite.value ← value
13:         return
14:     end if
15:             ▷ T fails to acquire ownership
16:     pointerWrite ← vbox.tentative
17:     orec ← pointerWrite.orec
18: end if
19: if orec.owner.root ≠ T.root then
20:             ▷ write-write conflict between top-level trans.
21:     ownedbyAnotherTree(T,vbox,value)
22:     return
23: end if
24: for all pointerWrite in vbox.tentative  do
25:             ▷ Tentative ver. list owned by another sub-txn
26:             ▷ T iterates over the list to insert its version
27:     if follows(T,pointerWrite.owner) then
28:         tentativeWrite.CASnext(new TentativeVersion(value))
29:         return
30:     end if
31:     if pointerWrite.orec.owner == T then
32:         pointerWrite.value ← value
33:         return
34:     end if
35: end for
```

tion acquired the ownership of the tentative write, in which case T must check if the new owner belongs to a different transactional tree by comparing the roots of the transaction trees. If the owner belongs to a different transaction tree, i.e., inter-tree conflict occurs, the transaction uses a fallback mechanism, *ownedbyAnotherTree()* (line 21). This mechanism aborts the sub-transaction, up to the root ancestor. Then it re-executes the af-

fected sub-transactions in the context of the root top-level transaction. The key difference is that top-level transactions maintain a traditional write-set, called *rootWriteSet*, to use when a VBox is locked by another transactional tree.

If the owner of the tentative version at the head of the list belongs to the same transactional tree (line 24), T iterates over the tentative version's list until it finds a proper place to insert the new tentative write. Recall that the tentative version list is organized by descending order of recency, according to the serialization order imposed by the strongly ordered semantics. In order to determine whether T should be serialized after the owner, say T', of a tentative version, the *follows()* functions is used, which compares the *ancVer* of T and T' and identifies their first (closest to the root) ancestor of T not in common with T', say T''. If T'' is a continuation, or if $T'' = \emptyset$ (i.e., T and T' have the same parent) and T is a continuation, then T follows T' in the serialization ($T' \rightarrow T$).

As we will discuss shortly, maintaining the tentative version list sorted by serialization order, allows better performance of the read procedure as reads can return the first value encountered in the *tentative write list* which meets the value visibility rule, avoiding iterating the whole list. Furthermore, maintaining the tentative version list sorted ensures that when the top-level transaction commits, it can find the values that must be written back to the permanent write lists on the head of the tentative write lists.

4.2.2 Read operations

The pseudo-code for managing the read operations of a sub-transaction is reported in Algorithm 4.2.

When processing a read, a transaction T checks whether the latest tentative write was performed by a transaction that has already committed or aborted (line 6). This means that this VBox does not contain tentative versions created by sub-transactions belonging to the same transactional tree of T. Hence, the read is simply served by selecting the most recent version in the permanent list committed before T started (this logic is encapsulated in the *readFromPermanent*() primitive).

Otherwise, T needs to check whether itself or any of its ancestors have previously written to the *VBox*. At this point, T iterates over the tentative version list of the *VBox* until one of the following conditions is verified: (1) T is the owner of the tentative version. In this case, the transaction also

Algorithm 4.2 Read Procedure by Transactional Future or Continuation

```
 1: Read(T,vbox):
 2: pointerWrite ← vbox.tentative
 3: orec ← pointerWrite.orec
 4: status ← orec.status
 5:         ▷ fast path when no read-after-write
 6: if status ≠ RUNNING then
 7:    return readFromPermanent(vbox)
 8: end if
 9: while pointerWrite ≠ null do
10:    if pointerWrite.owner == T ∧ status ≠ ABORTED then
11:       return pointerWrite.value
12:    end if
13:    if T.ancVer.contains(orec.owner)
       ∧ orec.txTreeVer ≤ T.ancVer.get(orec.owner) then
14:             ▷ add to the child transaction T's read-set
15:       T.readSet.put(vbox)
16:       return pointerWrite.value
17:    end if
18:    pointerWrite ← pointerWrite.previous
19:    orec ← pointerWrite.orec
20: end while
21:         ▷ confirm no read-after-write could happen
22: if T.root.writeset.contains(vbox) then
23:    return T.root.writeset.get(vbox)
24: end if
25: return readFromPermanent(vbox)
```

needs to make sure that the write does not belong to a previous aborted execution. This previous execution corresponds to the case in which the sub-transaction failed validation and was forced to re-execute. If the write was performed in T's current execution, then no further checks are needed and the procedure returns that value (line 11). (2) The owner of the tentative write is an ancestor of T. Under this circumstance, T may read that entry only if the entry was made visible by its owner before T started. This is enforced by looking up in the *ancVer* what is the maximum version of the ancestor's write the transaction can read and comparing it with the version of the tentative write, *txTreeVer* (line 13-19).

If no valid value was found in the tentative version list, then there are no tentative versions produced by sub-transactions in T's transactional tree that are visible to T. Therefore, T checks whether its top-level transaction

Algorithm 4.3 Wait commit rule used to enforce strongly orderedsemantics

```
1: WaitTurn(T):
2: if T is a Continuation then
3:     wait until T.parent.nClock == 1
4:     return
5: end if
6:         ▷ T is a Transactional Future
7: Set ancCont ={T' ∈ T.ancVer : T' is a continuation}
8: Transaction anc = closest ancestor of T in ancCont
9: if anc = ∅ then
10:    return
11: end if
12: wait until anc.parent.nClock==1
13: return
```

had already written to the same VBox (lines 21-22). If this is the case, it reads from the top-level transaction's write-set. Else, it fetches a committed value from the permanent version list.

4.2.3 Commit procedure

The pseudo-code for committing and aborting a sub-transaction is reported in Algorithm 4.4.

When a sub-transaction T tries to commit, it needs to validate its read-set in order to detect if it has missed the writes produced by other sub-transactions belonging to the same transactional tree and preceding T in the serialization order imposed by the strongly ordered semantics. To this end, before activating its validation procedure, a sub-transaction first waits for the commit events of every sub-transaction that should be serialized before it. We postpone shortly the description of the logic used by SO-JTF to enforce this serialization order, which is encapsulated in the *waitTurn* function.

The validation phase, encapsulated by the *validate*() primitive (line 3), scans the tentative version lists of the VBoxes read by the transaction. If a version is found belonging to an ancestor of T, and this version does not coincide with the one in T's readset, validation fails and the transaction must be aborted and re-started. If the transaction was running a transactional future, it simply calls the abort method and re-executes the transactional future from the beginning. Otherwise, if the transaction was running a

Algorithm 4.4 Commit and Abort Procedure by Transactional Future or Continuation

```
1: Commit(T):
2: waitTurn(T)
3: if ¬ validate(readSet) then
4:     Abort(T)
5:     return
6: end if
7: T.parent.nClock += 1
8: T.orec.txTreeVer ← parent.nClock
9: T.orec.owner ← parent
10: for all t in committedChildren  do
11:     t.orec.txTreeVer ← parent.nClock
12:     t.orec.owner ← parent
13: end for

14: Abort(T):
15: waitTurn(T)
16: for all vbox in T.boxesWritten do
17:     pointerWrite ← vbox.tentative
18:     if pointerWrite.orec.owner == T then
19:         revertOverwrite(vbox)
20:     end if
21: end for
22: T.orec.status ← ABORTED
23: for all t in childrenTransactions do
24:     t.orec.status ← ABORTED
25: end for
```

continuation, it aborts and uses the FCC support in order to restore the execution state to the point where the continuation started.

If a sub-transaction T passes the validation phase, it increases by one the *nClock* of its parent transaction (line 7). Next T propagates its own writeset, and that of its child transactions, to the parent (sub-)transaction. To this end, T updates its own *orec*, and the ones of its child transactions, by setting : i) the owner field to point to its parent transaction; ii) the *txTreeVer* field to *nClock*'s value of its parent transaction.

When aborting, transactions must revert the writes they performed when their write is at the head of the tentative version list. In this case, an aborting transaction cannot directly mark its tentative version as aborted. In fact, the head of the tentative list is used to establish a lock that grants

access to the entire transactional tree[2]. If some other sub-transaction of the same transactional tree had stored a version v in the second position of the tentative list, the head of the tentative version list has to be substituted with v. In order to make this operation lock-free, we need to make sure that no transaction changes any of the version between the tail of the list and the version of the aborting transaction [73]. To ensure this, a transaction T only aborts when every transaction that could precede T in the serialization order has finished executing (either aborted due to an inter-tree conflict or committed). This is done, just like for the case of commits, by using the *waitTurn*() function.

Finally, the transaction completes its abort by changing the status of the *orec*s it controls (its own *orec* and its children transactions' *orec*) to ABORTED (lines 22-25).

4.2.4 Transaction serialization order

To enforce a transaction serialization compliant with the strongly ordered semantics defined in Section 3.1.4, SO-JTF commits, at any point in time, at most one sub-transaction of a transactional tree, and only after having ensured that all the sub-transactions that precede it in the serialization order have already committed.

It should be noted that, since SO-JTF supports nesting of transactional futures, it is, in general, not possible to establish the serialization order of sub-transaction *a priori*, i.e., upon their creation. Consider, for instance, the transactional tree in Figure 4.1a: had T_{F1} not submitted T_{F2}, T_{C4} would have received a serialization order equal to 2 (and not 4).[3] SO-JTF tackles this problem by establishing the commit order of transactional futures, and continuations *a posteriori*, via two simple and lightweight waiting rules, one for transactional futures and one for continuations (see *waitTurn*() in Algorithm 4.3).

The waiting rule for a continuation, say T, is very simple: it suffices to wait till the *nClock* of its parent becomes equal to 1. This means that

[2]Recall that, when a transaction is performing a write, it checks if the owner of the write at the head of the list has already finished (line 10 of Algorithm 4.1). Once the lock is established, only transactions in the same transactional tree of the transaction are allowed to write to the list.

[3]On the other hand, if nesting of futures was not to be supported, futures would be submitted only by the thread executing the top-level transaction, and their serialization order would coincide with their submission order.

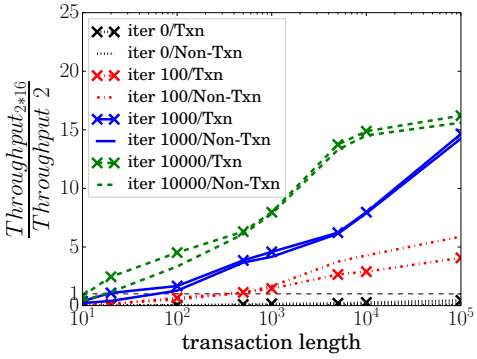
the sub-tree rooted in the corresponding transactional future, which directly precedes T in the serialization order, has already committed.

The waiting rule for a transactional future, say T_F, is slightly more complex, but it is also based on the same principle: identifying the transaction that immediately precedes T_F in the serialization order. Let T_C be the ancestor of the first continuation encountered by traversing "upwards" (from the closest to the furthest ancestor) T_F's *ancVer*. If $T_C = \emptyset$, then it means that T_F is the first transactional future to commit according to the strongly ordered semantics, and can commit without incurring any wait. Else, if such a T_C exists, it is safe for T_F to validate and commit as soon as the left-subtree of T_C's parent has completed execution. In fact, no other sub-transaction can, from that point on, ever commit and serialize before T_F. Hence, a transactional future can only commit when the *nClock* of T_C's parent is equal to 1.

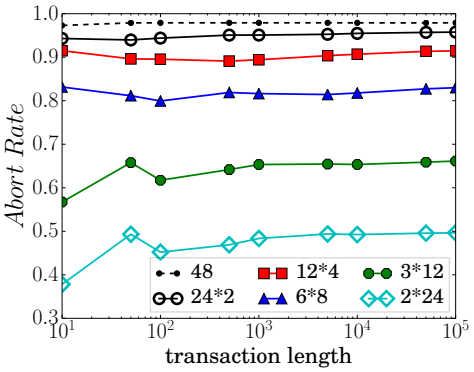
4.2.5 Optimizing read-only transactions

Since SO-JTF uses a multi-versioning concurrency control, the snapshot observed by read-only top-level transactions is guaranteed to be consistent (although possibly obsolete, i.e., not reflecting the effects of update transactions that committed during the read-only transaction's execution). Thanks to this property, read-only top-level transactions can skip validation, and immediately commit when they finish executing.

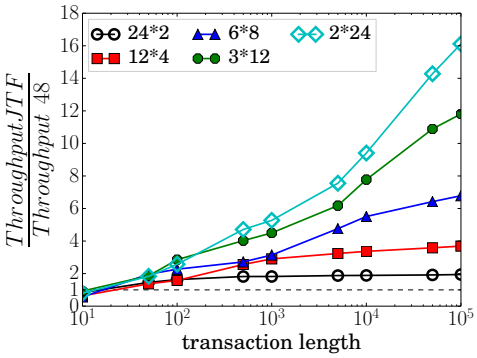
However, even if transactional futures were marked as read-only, it is in general not safe to skip their validation. In fact, a read-only transactional future may miss the write performed by a preceding (according to the serialization order) sub-transaction. On the other hand, in SO-JTF, we detect and take advantage of the situation in which it is actually safe to skip the validation of a read-only transactional future. This is true whenever all other sub-transactions that precede it in the serialization order have already committed before it started, or if all of those transactions are read-only as well. To support the latter optimization, we added a new list in every top-level transaction, which contains the identifier of every committed read-write sub-transaction of a transactional tree. Whenever a read-only transactional future reaches the commit phase, it waits for its commit turn using the *waitTurn*() primitive. Then it checks if there is any read-write sub-transaction in this list. If the list is empty, then the transaction can safely skip validation at the end of execution.



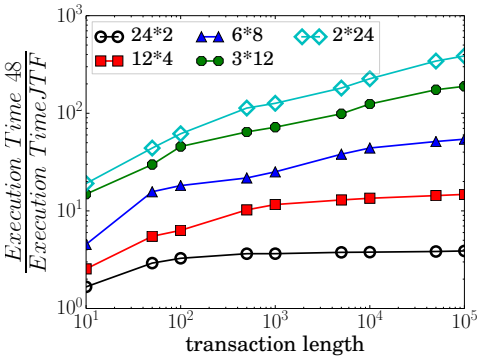
(a) Read-only workload: normalized throughput w.r.t. 2 top-level transactions.



(b) High Contention workload: abort rate.



(c) High Contention workload: normalized throughput w.r.t. 48 top-level transactions.



(d) High Contention workload: normalized transaction execution time w.r.t. 48 top-level transactions.

Figure 4.3: Synthetic Benchmarks.

4.3 Evaluation

In this section we conduct an experimental evaluation aimed to seek answers to the following key questions:

- How large are the overheads introduced by SO-JTF to coordinate the execution of transactional futures, and how do they compare with the inherent costs of *non-transactional* futures?
- What is the minimum granularity of a transaction for which it is beneficial, from a performance perspective, to use parallelization via transactional futures?
- What performance gains can be achieved by SO-JTF in presence of diverse workloads encompassing both synthetic benchmarks and well-known transactional benchmarks?

The results presented in this section are the average of five runs, executed on a machine with a AMD Opteron 6168 processors (48 cores total), 128GB of RAM, Linux 2.6.32 and an OpenJDK 64-bit Server 1.7.0 (build 19.0-b03) JVM.

Synthetic benchmarks. We start by presenting the results obtained by using a synthetic benchmark that we designed to exert tight control on the characteristics of the generated workload and to emulate extreme scenarios that allow us to assess the overheads and potential gains of SO-JTF. This benchmark generates, in each transaction, a configurable number of read and write memory accesses to an array of 1M elements. In between two memory accesses, the benchmarks emulates the execution of CPU-bound computations via a loop that executes a tunable number of iterations, called *iter*, in which we issue register-based arithmetic operations. Tuning the value of *iter* allows to generate diverse load pressure for the CPU and memory subsystems, and synthesize CPU-bound rather than memory-bound workloads.

We start by considering, in Figure 4.3a, a workload composed solely by read-only transactions, in which we vary the transaction length (i.e., number of read accesses within a transaction) from 10 to 100K and the number of CPU-bound iterations between two consecutive memory accesses from 0 to 10K. The items targeted by read operations are selected uniformly at random across the whole array. It should be noted that, since we consider a workload composed exclusively of read-only transactions, synchronization

is, in fact, unnecessary, and correctness could be ensured even by using a plain/non-transactional future implementation. Also, with such a conflict free workload, futures can provide no benefit by reducing the likelihood or the cost of aborts. Hence, by comparing the performance of SO-JTF with that of a non-transactional future in this workload, we can isolate and quantify the overhead induced to enforce the transactional future semantics and the costs that are inherent to the use of futures, such as inter-thread communication and increased contention on the memory bus among the threads executing the future(s) and the continuation. On the y-axis, we report the normalized throughput achievable by parallelizing the transactional code across 16 threads (i.e., 15 executing futures and 1 executing in the continuation) and running concurrently two top level transactions. The normalization is with respect to a baseline that does not exploit futures and uses two threads. Hence, the maximum throughput increase expectable amounts to 16×.

The data in Figure 4.3a shows that SO-JTF achieves close to optimal performance if transactions have at least 10K memory accesses and the workload contains sufficiently long CPU-bound computations. In fact, when setting the value *iter* to 0 (which yields a completely memory bound workload), increasing the degree of intra-transaction parallelism just hampers performance even for long-running transactions that execute 100K memory accesses. This is somewhat unsurprising, as by increasing the total number of concurrently active threads from 2 to 32, the memory subsystem is subject to a 16× larger load. In a memory-bound workload, this causes a surge in the average number of stall cycles incurred by CPUs due to memory accesses (a fact which we experimentally verified via the *perf* profiler). Yet, it is interesting to highlight that these overheads are not caused by the concurrency control scheme used in SO-JTF. In fact, Figure 4.3a shows that the performance of SO-JTF is quite close to that of a non-transactional future implementation. These data confirm that most of the overheads incurred by SO-JTF are inherent to the usage of futures, and that the additional overheads introduced by SO-JTF are modest even in such a challenging scenario (the average slowdown w.r.t. non-transactional future is <1%).

Next, we set *iter* to 1k and consider conflict prone workloads. In this case, transactions perform a variable length prefix of read accesses, followed by 10 update operations on a set of 20 "hot spot" items (selected uniformly at random with restitution). We report the normalized throughput achieved by SO-JTF when using a total of 48 threads, which we allocate to execute

either futures or top-level transactions/continuations. In Figure 4.3, we use the $i * j$ notation to indicate that we execute i top level transactions, each parallelized via j threads ($j - 1$ executing futures and one executing the continuation). We use as baseline for normalization a configuration in which we use 48 concurrent top-level transactions (i.e., no futures). This allows us to compare whether, given a fixed pool of available threads, it is more beneficial to exploit them in order to pursue inter-transaction or intra-transaction parallelism.

As shown in Figure 4.3b, in such a high contention workload, the usage of transactional futures significantly reduces the likelihood of conflict between transactions. This is due to two main factors: i) the more threads are allocated to futures, the smaller the number of concurrent top-level transactions, and, consequently, the lower the probability that the latter incur a conflict; ii) also, intra-transaction parallelism (via futures) allows for reducing the execution time of a single-top level transaction, which, in its turn, reduces the transaction vulnerability window as well as the cost of a transaction's restart. Such a reduction of the abort rate amplifies remarkably the gains achievable by SO-JTF, when compared to the previously considered read-only workload: the throughput gains (Figure 4.3c) start to be significant with much shorter transactions and the transaction execution latency, which accounts also for transactions' retries due to abort, is reduced by up to 400×. This is explainable considering that, in this high contention scenario, transactions can be re-executed tens of times before being committed, whereas the average number of transaction re-executions is around one in the 2*24 configuration.

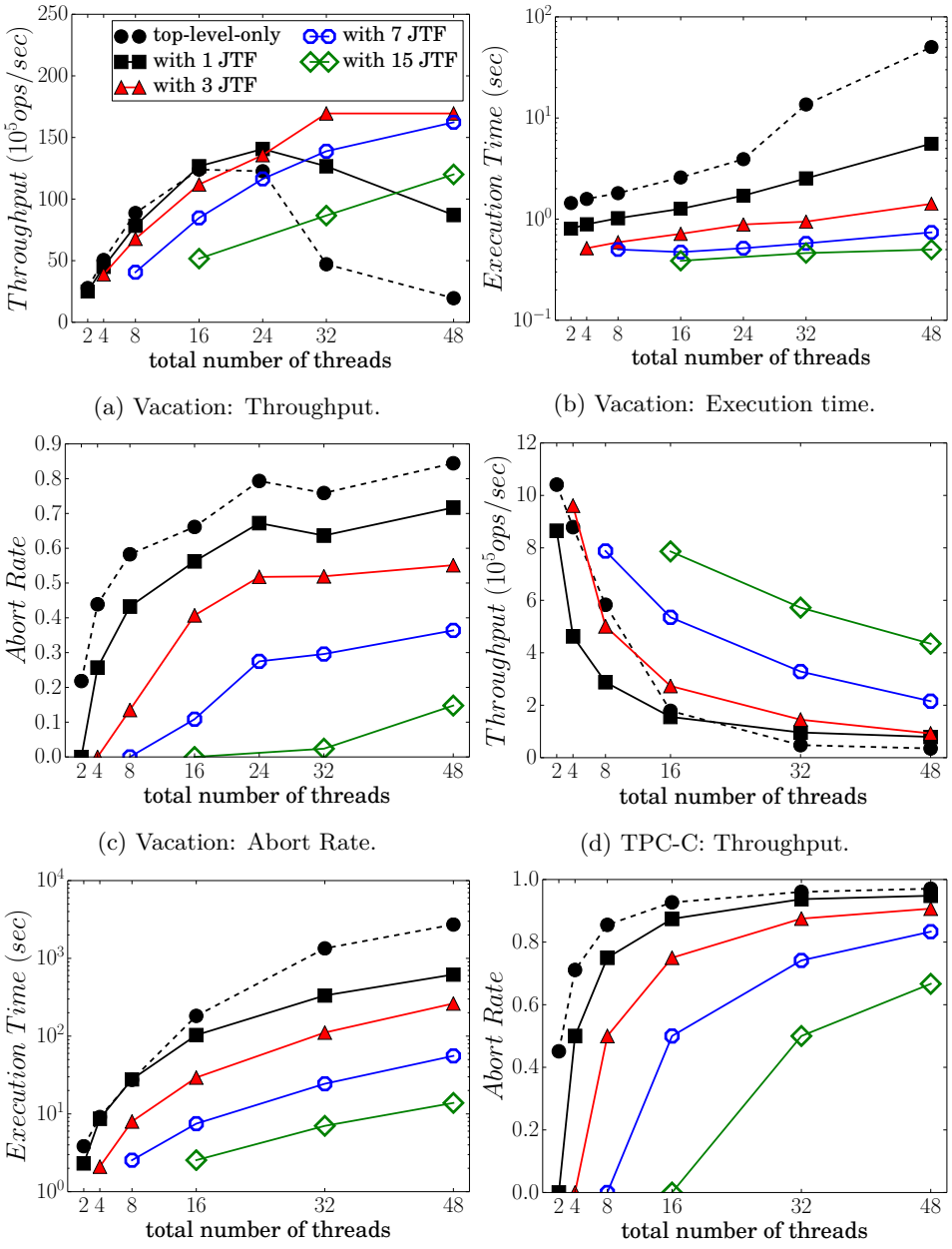
Vacation and TPC-C. We now consider two well known and more realistic benchmarks for transactional systems: Vacation of the STAMP benchmark's suite [92] and TPC-C [93]. The former is a popular benchmark for TM systems, which emulates the activities of a travel agency; the latter is an OLTP benchmark for database systems that mimics the activities of a warehouse supplier. We adapted these benchmarks to be parallelized using SO-JTF. In both benchmarks, we use transactional futures to parallelize long running transactions that execute a long cycle, during which they read a number of domain objects and compute various functions, e.g., they identify travels within a given price range, or compute the total amount of money raised by the warehouse.

For both the Vacation and the TPC-C benchmark we report in Fig-

ure 4.4 the average throughput, execution time and abort rate. We use as independent variable in this experiment the total number of active threads, and report data for five different thread allocation strategies: a baseline in which no transactional future is used, and all available threads execute independent top-level transactions; four alternative configurations, which allocate the available threads to parallelize top-level transactions by means of, respectively, 1, 3, 5, and 7 transactional futures (along with one continuation thread).

By comparing the throughputs achievable by the two benchmarks (see Figure 4.4a and Figure 4.4d), we observe that, if transactional futures are *not* used, Vacation can scale up to approximately 16 threads; TPC-C, on the other hand, generates an inherently non-scalable workload, for which using more than one top-level transaction causes a quick surge in the transaction conflict probability (see Figure 4.4f) and only hinders performance.

Overall, the use of transactional futures produces an improvement on performance, which is similar in both benchmarks. By allocating available threads to parallelize (a smaller number of) top-level transactions, rather than to activate additional, conflict-prone, top-level transactions, the likelihood of contention is strongly reduced. In other words, at high thread counts, the transactional future abstraction allows to utilize available computational resources in a much more effective way. From the throughput's perspective, transactional futures allow to extend the maximum achievable throughput by nearly 50% in Vacation and to scale up to 48 threads. In relative terms, the throughput gain when using 12 top-level transactions parallelized with 3 transactional futures (plus 1 continuation) is 8.4× larger than when using 48 non-parallelized top-level transactions. With TPC-C, where contention among top-level transactions is higher than in Vacation, the relative throughput gains deriving from transactional futures, at parity of totally used threads, are, unsurprisingly, even larger, extending up to 10.7× (again at 48 threads). Even more impressive are the benefits in terms of reduction of the time required to commit a transaction, see Figure 4.4b and Figure 4.4e, with gains peaks of up to two orders of magnitude for both benchmarks. Such striking gains can be explained, as already discussed when analyzing the synthetic benchmarks, by considering that transactional futures not only allow for reducing the average number of transaction restarts due to aborts, but also for reducing the cost incurred when aborting transactions (i.e., the mean execution time of aborted transactions).



(a) Vacation: Throughput.

(b) Vacation: Execution time.

(c) Vacation: Abort Rate.

(d) TPC-C: Throughput.

(e) TPC-C: Execution time.

(f) TPC-C: Abort rate.

Figure 4.4: Vacation and TPC-C Benchmarks..

4.4 Conclusion

In this chapter we presented SO-JTF, a Java-based implementation of transactional futures. The core of SO-JTF is an innovative concurrency control algorithm, which strives to maximize parallelism by using multi-versioning techniques, while enforcing strong guarantees regarding the serialization order of futures.

We evaluated SO-JTF by using synthetic benchmarks as well as two popular benchmarks for TM and database systems. The experimental results show that the use of futures allows not only to untap parallelism within transactions, but also to reduce the cost of conflicts among top-level transactions in high contention workloads.

Chapter 5

An implementation of the Weakly Ordered Transactional Future Abstraction

This chapter presents how to implement the proposed WO (Weakly Ordered) transactional semantic by introducing a software transactional memory (STM) algorithm that orchestrates the execution of transactional futures via a novel graph-based concurrency control scheme. We implement this algorithm by extending a state of the art multi-versioned STM (JVSTM [35, 52]), and quantify performance trade-offs achievable by adopting the proposed semantics via an experimental study encompassing several diverse workloads.

Some passages in this chapter have been quoted verbatim from [87].

5.1 Overview of the System

In this section we introduce WO-JTF (Weakly Ordered transactional futures), a STM that implements the weakly ordered semantics presented in Section 3.1.4.

WO-JTF has been developed by extending JVSTM [35, 52], a state of the art STM in Java that employs an efficient multi-versioning concurrency mechanism to regulate concurrency among top-level transactions. Yet, techniques used by WO-JTF to orchestrate the execution of transactional futures are largely orthogonal to the mechanisms used to regulate concurrency

among top-level transactions. Thus, in the following we will abstract the mechanisms used to regulate concurrency among top-level transactions, and simply assume the availability of a "conventional" STM implementation, i.e., that does not support intra-transaction parallelism, but ensures *opacity* [39] among concurrent top-level transactions.

We present WO-JTF in an incremental fashion, assuming initially that no future escapes from its spawning top-level transaction, and we discuss how to avoid this assumption in Section 5.1.2.

5.1.1 Base algorithm

WO-JTF maintains, for each top-level transaction T, a graph, noted $\mathcal{G}$, that is used to track the logical dependencies among the transactional futures spawned or evaluated by T, either directly or indirectly, i.e., via other transactional futures spawned by T. Unlike the FSG (see Section 3.1.4), $\mathcal{G}$ does not use bi-paths to encode *all* possible serialization orders of futures. Conversely, $\mathcal{G}$ uses only simple directed edges and, as such, is a plain directed graph and not a polygraph. This is done for efficiency reasons, given the inherent cost of managing polygraphs at run-time. As a consequence of this design choice, WO-JTF may reject schedules that are admissible according to the formalized semantics. This represents a classic trade-off in the design of concurrency control schemes [18, 94].

$\mathcal{G}$ is consulted in two main occasions. First, when one of the sub-transactions of T, say T_{sub}, reads a shared variable V, in order to establish which version of V should be observed. To determine the visibility of versions, the $\mathcal{G}$ of T is used to retrieve the "ancestors" of T_{sub}, i.e., the sub-transactions of T that are serialized before T_{sub}. The version observed by T_{sub} is the one included in the write-set of the closest ancestor of T_{sub}, if any, or the version visible by the top-level T according to the underlying STM, if no ancestor wrote to V.

Second, when one of the sub-transactions of T, say T_{sub}, requests to commit. In this case T_{sub} executes a validation scheme aimed at enforcing that T_{sub} can be serialized, either at submission or at evaluation. We will discuss the validation mechanism more in detail shortly.

Each vertex of $\mathcal{G}$ is associated with a sub-transaction and has a corresponding *status*, which is initialized to *active*, when the vertex is created, and is updated to *iCommit* (internally committed), when the sub-transaction issues COMMIT().

During the execution of a sub-transaction, i.e., in its *active* state, all the updates it produced are buffered privately, i.e., they are not visible to any other sub-transaction. When a sub-transaction T' enters the iCommit state, its updates are made visible to the other sub-transactions of the same top-level transaction, say T. The updates of all the sub-transactions of T will become atomically visible to other top-level transactions (and to their sub-transactions) only when T is committed.

$\mathcal{G}$ is initialized when a top-level transaction T starts with a single "root" vertex. Every time a *submit*(T') operation is issued to spawn a transactional future T' two new vertexes are created, one corresponding to the future and the other to its continuation. These two vertexes are connected through two edges that depart from the vertex of the (sub-)transaction that spawned T'. To ensure that the write-set of a sub-transaction, T_s, that spawns a transactional future, T_f, is visible to T_f (as T_f is serialized after T_s in $\mathcal{G}$), sub-transactions are automatically iCommitted whenever they issue *submit*().

When an *evaluate*(T') operation is issued, a vertex V is created and linked via two edges originated, respectively, on the vertexes associated with the future T' and its continuation.

Commit logic. When a sub-transaction T requests to commit, WO-JTF attempts first to serialize T at submission time, which implies accessing $\mathcal{G}$ in order to: i) merge the write-set of T with the write-set of the sub-transaction that spawned T; ii) remove T's vertex from $\mathcal{G}$.

If T cannot be serialized at submission, T is not aborted. Its vertex in $\mathcal{G}$ is marked as *completed* (but not *iCommitted*, so its updates are invisible) until some sub-transaction T' issues EVAL(T). At that point, T' will attempt to serialize T at its evaluation point, which implies an analogous manipulation of $\mathcal{G}$: i) adding the write-set of T to the write-set of the sub-transaction that evaluated T; ii) remove T' vertex.

WO-JTF employs two validation mechanisms to determine if a sub-transaction can be serialized upon submission or evaluation, which we call *forward* and *backward* validation.

- *Forward validation.* This mechanism is used to determine if a future can be serialized at submission time. Recall that whenever a sub-transaction T_s spawns a future T_f, T_s is automatically iCommitted. This guarantees that T_f always observe a snapshot that reflects the updates of its ancestors in $\mathcal{G}$. Serializing T_f at submission time, though, corresponds to moving the position of T_f's vertex in $\mathcal{G}$ and ordering it

before the vertex of T_f's continuation, say T_c. A sufficient condition to ensure that this reordering neither affects T_c, nor the sub-transactions serialized after T_c , is that none of these sub-transactions has read any of the variables updated by T_f. To test this condition, WO-JTF navigates $\mathcal{G}$ moving "forward" from T_c and checking, for any reachable sub-transaction T_{sub}, if the write-set of T_f intersects with the read-set of T_{sub}.

- *Backward validation.* This mechanism is used to determine if a future can be serialized at evaluation time. In this case, $\mathcal{G}$ is navigated backwards, starting from the vertex associated with the evaluating sub-transaction until reaching the sub-transaction that spawned T_f. All the sub-transactions along this path have executed concurrently with T_f and their writes were not visible to T_f (since they were not among the ancestors of T_f). As such, T_f can only be reordered after these sub-transactions if they did not update any variable that T_f read.

Synchronizing the access to $\mathcal{G}$. The graph $\mathcal{G}$ is manipulated concurrently by the sub-transactions of the same top-level transaction. WO-JTF regulates these concurrent accesses via a mix of lock-based and lock-free techniques:

- *Updates of $\mathcal{G}$* that occur when sub-transactions start/commit (and are relatively infrequent) are regulated a read-write lock, acquired in write mode. $\mathcal{G}$ is also associated with a timestamp that is increased whenever $\mathcal{G}$ is updated and serves as a version counter to ensure the atomicity of traversals of $\mathcal{G}$.

- *Validation operations* synchronize with concurrent committing transactions by acquiring the $\mathcal{G}$ lock in read-mode.

- *Read operations* require establishing the set of ancestors of a sub-transaction in $\mathcal{G}$ and need to synchronize with concurrent updates of $\mathcal{G}$. Since reads are typically more frequent that commit operations, we avoid acquiring the read-write lock and use a lock-free synchronization approach: $\mathcal{G}$'s timestamp is read before and after traversing the $\mathcal{G}$ to extract the ancestors' list, repeating the traversal if the timestamp changes due to a concurrent update.

5.1.2 Escaping Transactional Futures

Let us discuss how to extend the base algorithm to manage escaping futures with both LAC and GAC semantics.

- **Locally Atomic Continuations.** Ensuring these semantics implies guaranteeing that an escaping future can be serialized either at submission time or after any operation issued by its spawning top-level transaction. In order to ensure this property, whenever a top-level transaction T requests to commit, it needs to verify whether it spawned any future that is still active and has not been evaluated. In the latter case, T has to block until all such futures are committed.

- **Globally Atomic Continuations.** With these semantics, an escaping future is not bound to be serialized within its spawning top-level transaction. This spares top-level transactions from blocking if, at commit time, there is any uncommitted escaping future. On the downside, if an escaping future T_f requests to commit after its spawning top-level transaction T_s commits, T_f loses the opportunity to serialize upon submission (as T_s's updates may have been in the meanwhile observed by other committed top-level transactions) and is bound to be serialized upon evaluation, within the $\mathcal{G}$ of a different top-level transaction T_e. In order to determine if the execution of T_f is compatible with this new serialization order, it is necessary to guarantee that the state observed by T_f during its execution is consistent at evaluation time. This can be ensured by validating the read-set of T_f and checking if it is still up to date at evaluation time, i.e., considering the updates produced by all the top-level transactions serialized before T_e and by all the sub-transactions of T_e serialized before the vertex associated to the evaluation of T_f in T_e's $\mathcal{G}$.

5.2 Evaluation

This section reports the results of an experimental study that aims to contrast the performance achievable when using the weakly versus strongly ordered transactional future semantics defined in Section 3.1.4.

With this study, we seek answers to the following key questions:

1. What is the minimum transaction granularity for which it is using (WO) futures can lead to speed-ups? (Section 5.2.1)

2. How large is the overhead incurred by WO-JTF when compared to the overhead incurred when using strongly ordered transactional futures or non-transactional futures? (Section 5.2.2)

3. In which workloads does WO allow for performance gains with respect to SO and to applications that do not use futures to parallelize transactions? And how significant are these gains? (Section 5.2.3)

We compared the performance of WO-JTF with: i) JVSTM, which serves as a baseline not supporting futures; ii) SO-JTF, which, as discussed in Chapter 4, supports transactional futures with strongly ordered semantics and which we refer to as SO, in the following;.

The reported results are the average of five runs, executed on an Intel Xeon CPU E5-2660 v4 @ 2.00GHz, with 56 cores in total, 64GB of RAM, Ubuntu 16.04.6 LTS and an OpenJDK 64-bit Server 1.7.0(build 19.0-b03) JVM.

5.2.1 When to use (WO) transactional futures?

To control the workload's characteristics in a predictable way, we use a synthetic benchmark, that generates a configurable number of reads and writes memory accesses to an array of 1M elements from within each transaction. Additionally, in between two memory accesses, CPU-bound computations are emulated by spinning for a configurable amount of iterations, indicated as *iter*.

We start by considering a read-only workload. In Figure 5.1, we plot the throughput using 2 top-level transactions, each parallelized with 16 futures, normalized with regard to the throughput running 2 top-level transactions. As such, the theoretical maximum speedup is 16×. We vary the number of reading accesses from 10 to 100K on the x-axis. At the same time, we vary *iter* from 0 to 100K. The read location is selected uniformly at random across the whole array composed of 1 million elements. It is worth noting that it is of no-use to employ any concurrency control, since the workload is read-only. Thus, by comparing the performance of WO-JTF and non-transactional futures, one can isolate and quantify the overhead due to enforcing WO

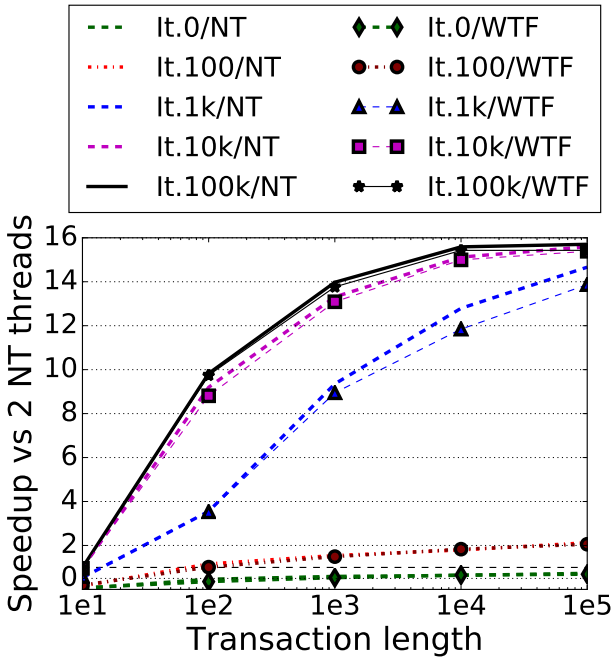


Figure 5.1: Speed-up of Weakly Ordered Transactional Future (WTF) (16 futures/2 top-level) versus 2 non-transactional (NT) threads in a read-only workload.

semantics as well as the inherent costs of using futures (e.g., inter-thread communication).

From Figure 5.1, we observe that WO-JTF achieves close to ideal speedups when transactions are sufficiently long and contain CPU-bound computations (*iter*>1000). When we use an fully memory-bound workload by setting *iter* to 0, increasing the degree of intra-transaction parallelism leads to negligible speedup even for long-running transactions that execute 100K memory accesses. This happens for both transactional and non-transactional futures, which suggests that the bottleneck is not the concurrency control logic of WO-JTF. Figure 5.1 also confirms that the performance of WO-JTF and of a non-transactional future implementation are quite close. This indicates that most of the overhead incurred by WO-JTF are inherent to the usage of futures and that scalability is not being limited by WO-JTF.

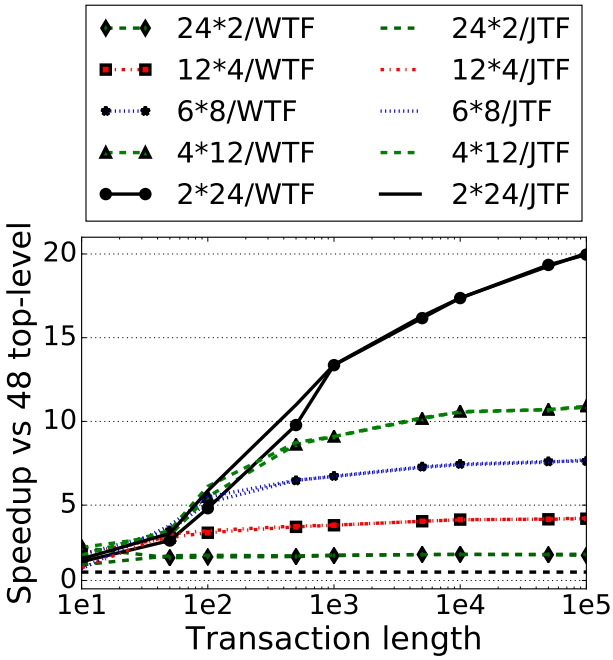


Figure 5.2: Speedup of Weakly Ordered Transactional Future (WTF)/ Strongly Ordered Transactional Future (JTF) with different top-level*futures w.r.t. 48 top-level in a contended workload.

5.2.2 Quantifying the overhead of WO-JTF with respect to SO-JTF

To quantify the overhead of WO-JTF w.r.t. to SO-JTF, we used a conflict-prone workload where WO-JTF can neither avoid aborts by serializing futures at evaluation nor benefit from avoiding stragglers. Specifically, futures execute a sequence of uniformly distributed read accesses over an array of 1 million elements, followed by ten update operations chosen uniformly at random from a (different) set of 20 "hot spot" items. Based on the findings of the last section, we set *iter* to 1k, to generate a workload where future-based parallelization can be beneficial.

We report the normalized throughput achieved by WO-JTF and SO-JTF when using a fixed total number of 48 threads. We allocate the 48 threads either to execute futures, continuations, or top-level transactions.

The results are reported in Figure 5.2. The notation $i * j$ means that we execute i top-level transactions, each parallelized via j threads. The result is normalized to a baseline, where we use all the 48 threads to execute top-level transactions concurrently. This setting helps to assess whether it is more beneficial to exploit inter-transaction or intra-transaction parallelism (using either WO-JTF or SO-JTF) given a fixed number of available threads.

In this conflict-prone workload, the use of transactional futures, either with weakly or strongly ordered semantics, reduces significantly the likelihood of conflicts between transactions as well as the cost of restarts, hence their speed-ups w.r.t. JVSTM. Note also that the performance of WO-JTF and SO-JTF are almost indistinguishable, confirming that WO-JTF introduces very limited overhead with respect to SO-JTF. The only exception is represented by the scenario of 2 top-level threads and 24 futures, where WO-JTF exhibits 10%-20% overhead when the length of read is below 500. We argue that this is due to the cost of synchronizing the manipulations of the graph structure used by WO-JTF to support WO semantics.

5.2.3 Quantifying the gains of WO-JTF with respect to SO-JTF

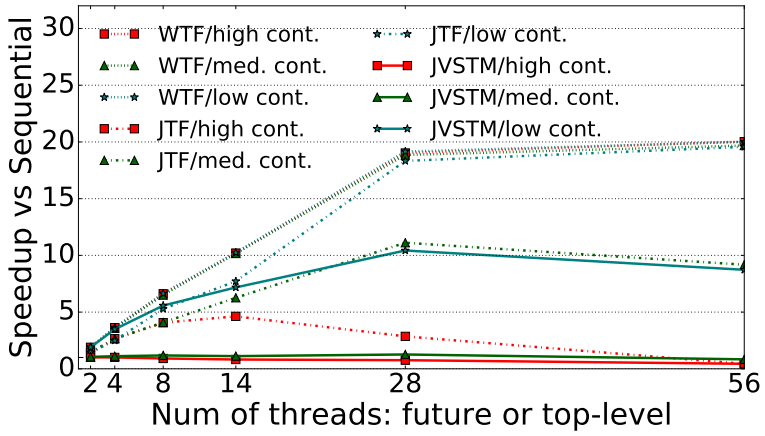
After showing that WO-JTF does not incur significant overhead w.r.t. to SO-JTF, we consider workloads where WO-JTF can outperform SO-JTF by either avoiding unnecessary aborts and/or avoiding stragglers. To this end, we conducted three experiments using three different workloads.

Synthetic Benchmark. We start by considering a workload where futures can conflict with their continuation. This causes the continuation to be aborted with SO semantics, whereas with WO the continuation's abort can be avoided by serializing its future upon evaluation. Each future first performs 10K reads from an array of 1M elements. Then, it writes once to a number of randomly selected hot spots.. Each continuation reads a random element from the set of the hot-spots and spawns a new future, until a given number of concurrent futures is reached. At that point, the top-level transaction evaluates all the futures it spawned (in spawning order) and commits.

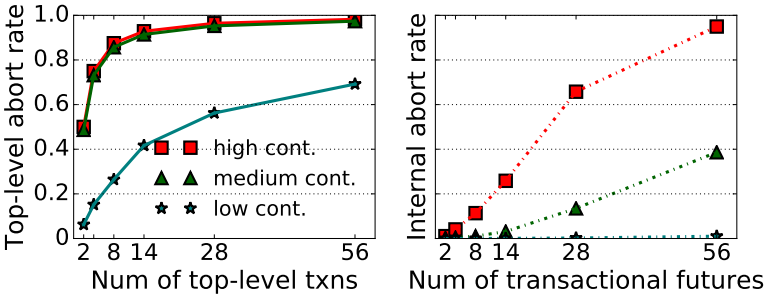
We set *iter* to 1k and vary contention by varying the hot spots' size: 100, 1K and 50K items.

In Figure 5.3a we report, on the y-axis, throughput normalized w.r.t. executing top-level transactions (no futures) sequentially.

We use the number of threads indicated on the x-axis as the number of



(a) Speedup using multiple futures (WTF(Weakly Ordered Transactional Future), JTF(Strongly Ordered Transactional Future)) or top-levels (JVSTM) w.r.t. 1 top-level (JVSTM).



(b) Abort rate using JVSTM (left) or SO-JTF (right).

Figure 5.3: Speedup (a) and abort rate (b) with different intra- and inter-transaction parallelism levels.

concurrently spawned futures for SO-JTF and WO-JTF, and as the number of concurrent top-level transactions for JVSTM. The worst performing baseline is the one that does not use futures, which incurs in the largest abort rates, see Figure 5.3b (left). In fact, when futures are not exploited to parallelize top-level transactions, these last longer and are, as such, more prone to conflict.

The throughput of SO-JTF is strongly affected by the degree of contention. In the high contention scenario, SO-JTF performance drops beyond

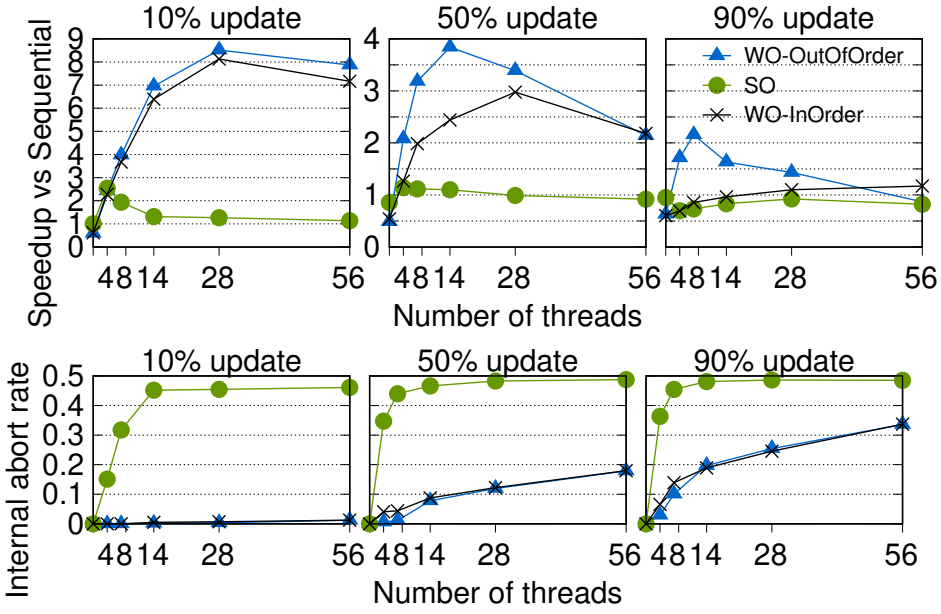


Figure 5.4: Throughput (top) and internal abort rate (bottom) for the Bank benchmark with 10%, 50% and 90% update operations, with Weakly Ordered Semantic Out of Order Evaluation (WO-OutOfOrder), Strongly Ordered Semantic (SO), and Weakly Ordered Semantic In Order Evaluation (WO-InOrder).

14 threads. This is explicable by analyzing the abort rate of futures and continuations, shown in Figure 5.3b (right). Conversely, the performance of WO-JTF is unaffected by the contention level since the WO semantic allows for serializing futures upon evaluation, sparing them from *any* abort. As a result, WO-JTF achieves peak gains of up to 20× (high contention, 56 threads) w.r.t. SO-JTF.

Finally, we evaluate the performance of WO-JTF using a benchmark, which we call *Bank*, that emulates replaying a log of operations from the daily records of a bank agency for backup or verification purposes. A similar benchmark has been frequently used to evaluate TM systems, e.g., [95–98]. This workload consists of two operations *transfer* and *getTotalAmount*. As the name suggests, *transfer* moves money from a list of sending accounts to a list of receiving accounts, while *getTotalAmount* returns the total bal-

ance of all accounts within the bank. As all transfers are between accounts belonging to the same bank, *getTotalAmount* is expected to always return the same quantity and serves as a sanity check to detect errors during the backup/verification process.

We set the total number of accounts of the emulated bank to 100K and fix the number of (pairs of) accounts involved by *transfer* operations to 100, selected uniformly at random. Also in this case, we set *iter* to 1000. Note that, in these settings, *transfer* operations are significantly shorter than *getTotalAmount* operations — thus the latter operations are prone to straggle the former ones.

To parallelize this workload using transactions (without futures), we divide the log of operations to be replayed into fixed chunks. Each chunk is executed sequentially using a top-level transaction, and we vary the number of top-level transactions that execute concurrently.

When using futures, instead of executing a chunk sequentially, each operation in the log is delegated to a future spawned from within the top-level transaction. We compare two variants of WO-JTF: *(i)* one that evaluates futures in the order they are spawned (WO-JTF-InOrder); and *(ii)* one that evaluates futures as soon as they complete execution (WO-JTF-OutOfOrder). The latter variant allows us to quantify the benefits of WO-JTF from avoiding straggling futures, as illustrated in Figure 3.3.

Figure 5.4 shows the speedup with respect to a sequential version and the internal abort rates for three workloads encompassing 10%, 50%, and 90% *transfer* operations (the remaining ones being *getTotalAmount*). From the plots, we can see that both WO-JTF variants outperform SO-JTF in all workloads, achieving up to ∼9× higher throughput. The abort plots show that WO-JTF variants incur significantly lower abort rates. In this workload, both *transfer* and *getTotalAmount* commute, therefore WO-JTF benefits from its ability to order futures at different serialization points, reducing abort rates. However, as the percentage of update operations increase, the aborts incurred by WO-JTF increase, lowering its speedup.

Finally, when comparing both variants of WO-JTF, we notice that evaluating futures out of order (to reduce the effect of stragglers) is always beneficial. As expectable, the largest gains stemming from straggling avoidance (larger than 2×) are achieved in the 50% and 90% update scenarios, as in the 10% update scenario the slowest operations (i.e., *getTotalAmount*) are by far the most common operation executed by futures — thus limiting the actual straggling effect.

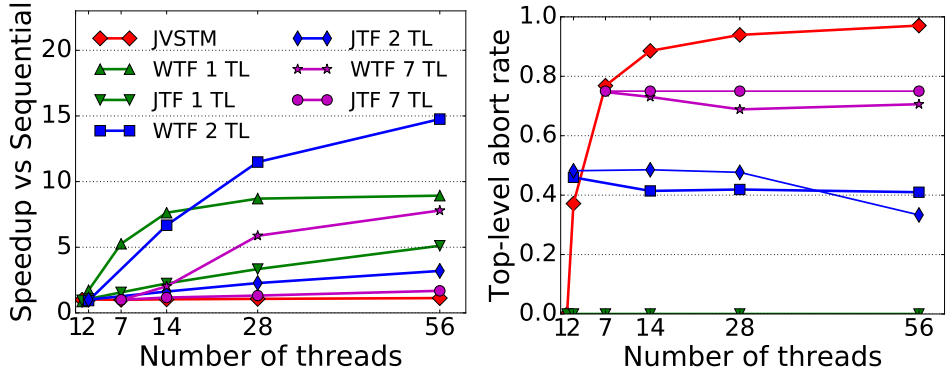


Figure 5.5: Vacation Benchmark: Speedup w.r.t. 1 top-level (left) and top-level abort rate (right) using different levels of intra- and inter-transaction parallelism, using top-level transactions(JVSTM), as well as Weakly Ordered Semantics(WTF), and Strongly Ordered Semantics(JTF) with various top-level transactions(TL).

Vacation Benchmark. Let us now evaluate the performance of WO-JTF using the Vacation benchmark of the STAMP suite [92], which we adapted to make use of transactional futures. Vacation emulates a travel agency and we parallelized its *MakeReservation* transaction using futures, similarly to what was done in previous work [73, 84, 99]: each reservation consists of a fixed number of search operations within a database of flights, cars, and hotels; we divide these search operations among a fixed number of futures and emulate the scenario in which some futures may have to access a remote database by injecting with 10% probability a delay of 100msec right after beginning a future.

Figure 5.5 (left plot) shows speedups of WO-JTF and SO-JTF and multiple top-level transactions (JVSTM) with respect to using a single top-level transaction. The x-axis denotes the total degree of parallelism that is allowed for each of these baselines, i.e., the number of active futures (set to 1 for JVSTM) $\times$ the number of concurrent top-level transactions. For SO-JTF, which only allows evaluating futures in the order in which they were spawned, a new future is activated as soon as the oldest spawned future completes; conversely, since WO-JTF supports out-of-order evaluation of futures, a new future can be spawned as soon as *any* future completes.

We can see that WO-JTF outperforms all baselines achieving up to $\sim$12$\times$

and ∼4.5× speedups compared to JVSTM and SO-JTF, respectively. The key reason for the gains of WO-JTF (and SO-JTF) with respect to JVSTM (see Figure 5.5, right plot) is the high probability of contention between top-level transactions and the high toll (waster work) imposed by the abort of top-level transactions. Since in this workload, there exists no conflict between futures, WO-JTF and SO-JTF achieve a similar abort rate, which is significantly lower than with JVSTM (as futures reduce the duration of each top-level transaction, reducing both the cost and chance of aborting). However, WO-JTF scales much better than SO-JTF, thanks to WO-JTF's ability to evaluate futures out of order that allows for mitigating the bottlenecks caused by straggling futures.

5.3 Conclusion

The chapter has described WO-JTF, a STM algorithm that combines multi-versioning and graph-based concurrency control techniques. We integrated WO-JTF with a state of the art JAVA-based STM (JVSTM) and evaluated its performance across a number of diverse workloads. Our experimental results allowed us to identify the scenarios in which transactional futures bring benefits and to quantify the performance tradeoffs that the different semantics offer.

Chapter 6

Online Tuning of Parallelism Degree

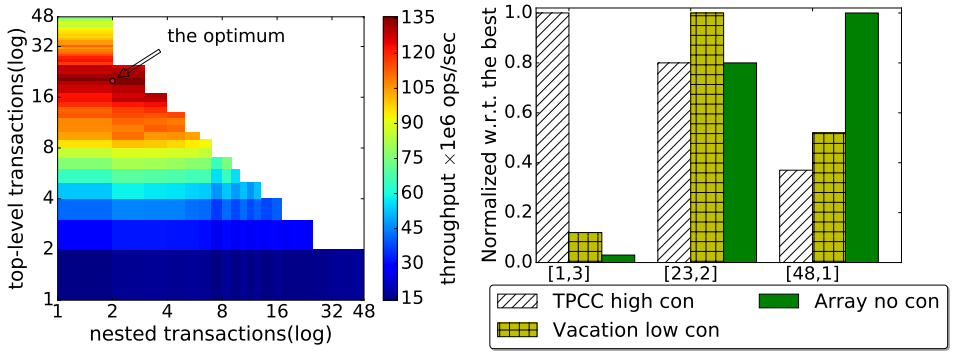
In this chapter, we address the problem of how to determine the optimal parallelism degree in TM systems that support intra-transactional parallelism, e.g., via transactional futures or conventional parallel nesting. The problem of identifying the optimal degree of parallelism has been long investigated in the TM literature. However, existing works have only considered TM systems with *no* support for intra-transaction parallelism. The problem complexity is inherently exacerbated in the presence of intra-transactional parallelism, where one has to identify the optimal parallelism degree not only for top-level transactions but also for nested sub-transactions. The increase of the problem dimensionality raises new challenges (e.g., increase of the search space, and proneness to suffer from local maxima).

Some passages in this chapter have been quoted verbatim from [100],

6.1 Background and System Model

A particularly challenging scenario for TMs is when workloads contain a large portion of long-running transactions. Indeed, long-running transactions suffer from long windows of vulnerability (i.e., the interval of time during which the transaction can be subject to conflicts with other transactions and abort), which makes them prone to prohibitively high abort rates.

A possible path to tackle the performance issues caused by long transactions is to reduce the number of concurrent transactions, while dividing



(a) Throughput of a TPC-C porting.

(b) No-one-size-fits-all config exists.

Figure 6.1: Performance of different configurations of inter- and intra-transaction concurrency in a PN-STM (Transactional Memory systems that support Parallel Nesting or transactional futures).

these fewer transactions into smaller, nested sub-transactions to run in parallel in the available idle cores, thus, reducing the execution time, and vulnerability window, of the original top-level transactions. This approach has received significant attention in the literature, with a number of proposals aimed to extend TM implementations to support parallel nesting with low overheads [69–73, 101].

Unfortunately, though, fully exploiting the potential of PN-STM (Transactional Memory systems that support Parallel Nesting or transactional futures) requires tackling a nontrivial problem that did not arise in TMs not supporting parallel nesting: identifying the right balance between inter-transaction and intra-transaction parallelism. Specifically, given a set n of available cores, in a PN-STM it is necessary to decide how many root (or top-level) transactions are allowed to be simultaneously active (t), and how many should be allocated to child (or nested) transactions (c) within each top-level transaction.

In CPU-intensive applications, as it is typically the case for TM environments, one should choose t and c such that the system is not oversubscribed, (i.e., $t \times c \leq n$). In platforms where t and c can be sufficiently high, there will be many possible configurations that avoid over-subscription: the search space grows in fact quadratically with respect to the problem of tuning the parallelism degree in TMs not supporting parallel nesting, which has been

subject of intense study in the literature [7–11].

Identifying the optimal configuration in the resulting bi-dimensional search space is far from trivial. Among the different configurations, it is necessary to decide whether to favor higher root parallelism while reducing nested parallelism, or vice-versa. This depends on complex, workload-dependent interference between root and child transactions that lead to contention and, therefore, conflicts; as well as on the overheads of spawning and maintaining nested transactions. Further, in contention-prone workloads, it is also not guaranteed that the optimal solution is the one that maximizes core utilization.

Figure 6.1a illustrates this optimization problem by presenting the throughput of a porting of the TPC-C benchmark executing on a PN-STM environment (JVSTM[73]), when varying t and c. For this particular workload, configuration (20,2), i.e., 20 top-level transactions, each with 2 nested transactions, will generate the best throughput, where the total number of cores in the system is 48. Furthermore, the throughput of the best configuration is 9× higher than the throughput of the worst, which is configuration (1,1), and 2× to 3× higher than that of most of the remaining configurations. We can also observe frequent local optima, which limit the effectiveness of solutions based purely on local search — which, instead, were shown to be effective in TMs that do not support parallel nesting[81]. Also, the best configuration for a given workload may be the worst for a different application, as illustrated in Figure 6.1b. Additionally, see Section 6.6.1, the static configuration that performs best on average across multiple workloads can be up to 3× away from optimum in some cases.

This is, to the best of our knowledge, the first work to address the problem of self-tuning the parallelism degree in PN-TM and propose a novel self-tuning mechanism, called AUTOPN (Automatic Parallel-Nesting). Due to the large search space of parallel-nested configurations, the key feature of AUTOPN is its ability to quickly prune regions in the search space of that are unlikely to contain useful configurations. As we show later, AUTOPN dramatically outperforms popular search heuristics (like gradient descent, simulated annealing, or genetic algorithms[10, 80, 102, 103]), which, in contrast to AUTOPN, tend to get trapped in local optima or avoid local optima at the cost of a large number of random searches.

In contrast, AUTOPN employs a novel combination of model-driven and local-search techniques to find optimal configurations by only searching a small subset of the search space, accurately and reactively. Since AUTOPN

requires no initial off-line training, it spares the cost of having to bootstrap an initial knowledge base representative of the target application and architectural environment.

Overall, this chapter of the dissertation makes the following contributions:

1. For an effective pruning of the search space of parallel-nested configurations, autoPN leverages an innovative design of an online learning approach tailored to PN-TM by combining model-driven and local-search techniques. In our approach, the exploration is driven by a regression model based on the lightweight M5P decision tree algorithm [12], which uses the feedback gathered during previous explorations. The output of this regression model drives a Sequential Model-based Bayesian Optimization process [13], which exploits the Expected Improvement (EI) theory in order to: (1) identify the most promising configurations to explore and (2) determine when to stop exploring (stopping criterion). Finally, the model-driven exploration phase is complemented by a refinement phase, using a local search based on a simple hill-climbing strategy.

2. Since the above mechanism is highly sensitive to a proper tuning of the duration of the feedback-monitoring windows, we propose a novel, PN-TM-specific, adaptive sampling heuristics to address this key configuration issue. Our heuristics aim at an optimal trade-off between measurement accuracy, reactiveness and convergence speed.

3. We integrated autoPN with JVSTM[35, 73], a state of the art Java library implementing a lock-free multi-version PN-STM. This allowed us to experimentally evaluate the proposed solution using both synthetic and standard benchmarks (Vacation of the STAMP benchmark suite [92] and a porting of the TPC-C benchmark) and compared autoPN versus five different general purpose online self-tuning approaches. On average, autoPN reaches stability 4× faster than its counterparts and converges to solutions that are, on average, less than 1% away from optimum.

6.2 Problem Definition

In this section, we first present some background information on PN-STM systems and introduce the abstract model of a PN-STM system that is considered by autoPN. Using this model, we propose a mathematical formalization of the problem of self-tuning the parallelism degree in PN-TMs.

6.2.1 PN-TM: Background and System Model

Let us start by introducing some base terminology related to the nested transaction model [104]: A transaction is either top-level, which is managed as a conventional non-nested transaction, or is nested within a (parent) top-level transaction. This nesting generates trees of transactions, which are denoted using family relationship terminology, such as parent, child, ancestor, descendant, sibling, etc. In this model, only transactions with no active children can access data; such accesses are either reads or writes to memory.

In the TM domain, various nesting models have been considered, defining different admissible behaviors regarding the concurrent execution of nested transactions[101]. A first distinction can be seen between flat (or linear) and parallel nesting, where the former supports only a single thread of execution in a transactional nest (i.e., it disallows concurrency within a transaction) and the latter allows actual concurrency among a parent and its children transactions. Another distinction regards the semantics associated with the commits of a nested transaction, which can be made applied at the "top-level" (i.e., made visible to other nests) either immediately (open nesting model) or only upon commit of it top-level transaction (closed nesting model).

The self-tuning scheme presented in this work is based on a black-box approach and, therefore, makes no assumptions on the commit semantics of nested transactions. As a matter of fact, AUTOPN is currently integrated with a PN-STM supporting the closed nesting model (JVSTM[73]), but it could be seamless applied also to open nested PN-STMs. Indeed, we consider an abstract model of a PN-STM, which assumes the following policy regarding how threads are allocated to top-level and nested transactions: top-level transactions are executed by a first set of threads, T; child transactions (of any family) share a disjoint set of threads P. Further, we assume that the PN-STM can alter the cardinality of sets T and P at run-time. Finally, we assume that the PN-STM system can be queried to obtain the measurement of some target key performance indicator (e.g., throughput) achieved in the current system's configuration.

This abstract model maps to various possible implementations, although, in typical PN-STM systems [70, 73], top-level transactions are often executed directly by application-level threads, whereas child transactions are executed by a shared thread pool that is under the direct control of the PN-STM run-

time (in order to minimize the overheads associated with the activation of child transactions).

6.2.2 Problem Formulation

In general, in a PN-TM program, each of top-level transaction may generate arbitrary deep trees of nested transactions, where each nested transaction may spawn a different number of child transactions. The shape of a transaction tree can also be dependent on the transaction type and on its inputs.

In the light of these considerations, one may be tempted to define the problem of tuning the degree of parallelism in PN-STM systems as aimed to identify the optimal (performance-wise) number of concurrent child transactions that should be executed for each parent transaction, at any depth of the tree, and for each different transaction type and input class. Although one may be allured by the generality of such a formulation, it is also easy to see that the corresponding problem's dimensionality is theoretically unbounded, and may thus make the problem intractable in practice.

In this section, we take a pragmatical approach, which leverages on practical considerations to reduce the problem's dimensionality and allow its tractability in realistic settings.

Observation 1: Transaction trees are normally shallow. The first observation that we make is that the PN-TM workloads considered so far in the literature[70, 71, 73] tend to generate very shallow, although potentially quite fat, trees. We argue that this is due to the fact that, in order to actually benefit from the use of parallel nesting, the tasks to be parallelized have to be sufficiently coarse in order to outweigh the overheads incurred for synchronizing the parallel execution child transactions. However, as we descend at greater depths in the tree, the task granularity tends to decrease exponentially, and, as such, also the probability of being able to effectively parallelize them.

Observation 2: Oversubscription should be avoided. A second pragmatical consideration is that in CPU-intensive workloads, such as those typically targeted by PN-TM, it is pointless to oversubscribe the available physical cores (or hardware threads, if simultaneous multi-threading is supported), i.e., it is desirable to ensure that the number of concurrent threads never exceeds the number n of physical cores.

Observation 3: Allocation of resources to top-level transactions

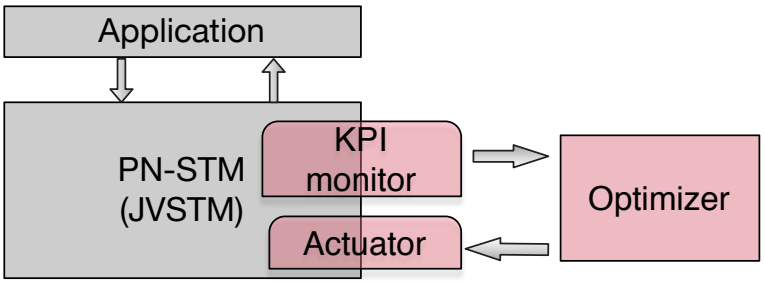


Figure 6.2: High level architecture of AUTOPN

should be fair. Finally, in most of the workloads that we are aware of, both in the flat and parallel nesting TM model [92, 105], the top-level transactions are executed with the same probability by any of the top-level threads in T. Hence, it is in general desirable to ensure a fair allocation of resources to top-level threads, i.e., each top-level thread should have the possibility to activate the same number of concurrent nested transactions. Achieving this goal, while still avoiding over-subscription, implies that each top-level thread can execute concurrently at most $n/|T|$ nested transactions.

Overall, the above considerations lead us to propose a more pragmatic problem formulation that considers a two dimensional search space $S = \{t \times c : t, c \in \mathbb{Z}^+ \wedge t \times c \leq n\}$, where t denotes the number of concurrent top-level transactions, c the number of concurrent nested transactions in each transaction tree, and n the total number of cores in the system. Denoting with $f : S \rightarrow \mathbb{R}$ the unknown function that maps the space of admissible configurations to their performance (quantified via some target Key Performance Indicator, i.e., KPI), the problem consists in identifying the configuration $opt \in S$ that optimizes f, i.e., maximizes or minimizes it depending on the target KPI. In the following, we will consider the problem of maximizing throughput (committed transactions per second), as target KPI.

6.3 Architectural Overview of autoPN

Figure 6.2 illustrates the high-level architecture of AUTOPN. AUTOPN has been integrated with JVSTM[73], a Java-based multi-versioned PN-STM supporting closed nesting. JVSTM was extended with three new modules,

namely, the *optimizer*, the *actuator*, and the *monitor*.

The *optimizer*, described in Section 6.4, aims to identify the optimal degree of parallelism for a given application workload. This component drives the exploration of the solution space by gathering feedback on the quality of the explored configurations via the KPI monitor, and requesting adjustments of the current degree of parallelism to the actuator. We based the implementation of the optimizer on Weka, an open source machine learning workbench in Java [106].

The *actuator*, described in Section 6.5, is the module in charge of steering the dynamic reconfiguration process of the degree of parallelism of JVSTM-based applications.

The *monitor*, described in Section 6.5, is responsible for gathering online measurements of relevant key performance indicators (KPIs) of the PN-STM system. As already mentioned, in the following we will focus on the problem of maximizing throughput, although AUTOPN could be used to optimize different metrics (e.g., latency or abort rate). The key challenge in the design of this component consists in gathering measurements in a both accurate and timely way, so as to maximize not only the accuracy but also the reactivity of the self-tuning process.

6.4 Optimizer

Figure 6.3 illustrates the self-tuning process that is coordinated by the optimizer module. The optimizer leverages two complementary methodological approaches: i) an initial model-driven exploration phase, which relies on the Sequential Model-based Bayesian Optimization (SMBO) framework [107] and on decision tree-based regression models [12]; ii) a final localized search based on hill-climbing. The rationale underlying this approach is to take the best of the two approaches (i and ii) in order to compensate for each other's drawbacks.

By steering the initial exploration using model-driven techniques, AUTOPN aims at building a global view of the unknown function, f, that maps the configuration space (S) to system's performance (e.g., throughput). This approach allows to quickly discard large regions of the search space that are unlikely to contain high-quality solutions. Model-based techniques, however, are inherently approximated (being based on approximate models built based on partial knowledge of S) and, although they are nor-

mally effective in identifying at broad strokes the quality of macro-regions of the search space, they tend to suffer from *long-sightedness*, i.e., they have low accuracy in predicting the quality of solutions in nearby search regions. This makes model-based techniques prone to quickly identify solutions that are in the proximity of global maxima, but fail to detect higher quality solutions in their proximity.

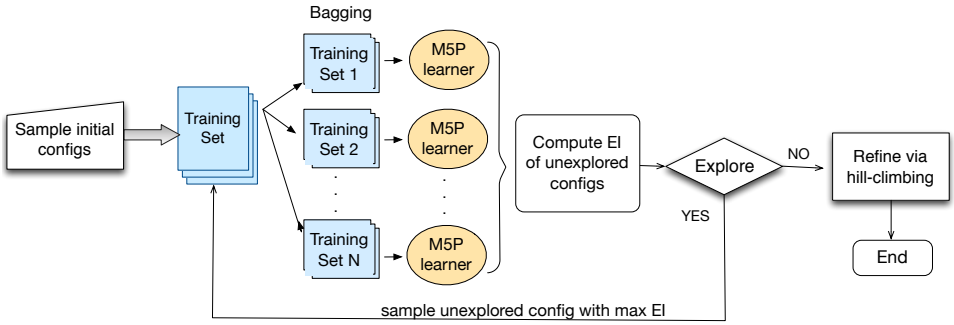


Figure 6.3: Illustrating the self-tuning process coordinated by the optimizer.

Local search heuristics, conversely, can suffer from *short-sightedness*, in the sense that they get trapped easily in local maxima, and, as we will show in Section 6.6, tend to exhibit poor accuracy when used alone. Conversely, in autoPN, we use a localized search, based on hill-climbing, only after having concluded the model-driven optimization phase. This way, we activate the local search starting from a high-quality configuration, which we strive to further refine via a final hill-climbing phase that addresses long-sightedness of the model.

6.4.1 Initial sampling

Any black-box model-driven approach requires a collection of initial samples of the configuration space aimed to construct an initial knowledge base/-training set to build the model. The most common approach to determine the initial training set consists in using a randomized, uniform sampling policy. The key advantage of this approach is its simplicity and generality.

In autoPN, we depart from this conventional design and exploit domain knowledge to define a biased sampling strategy that aims at promoting the construction of models able to capture the global trends of f based on a small number of configurations. The key intuition is to force the determinis-

tic exploration of 9 configurations that lie on the three boundary regions of S illustrated in Figure 6.4. The rationale underlying this strategy is to assess the workload's sensitivity to small variations of the inter-/intra-transaction parallelism in proximity of three "pivot" configurations, which correspond to three extreme settings of inter-/intra-transaction concurrency, namely: i) using only a thread to execute top-level transactions and disabling parallel nesting, i.e., (1,1); ii) allocating all hardware resources to top-level transactions and disabling parallel nesting, i.e., (n,1), iii) running top level transactions sequentially and allocating available cores to execute nested transactions concurrently, i.e., (1,n).

In Section 6.6 we will show that this biased, domain-specific, sampling strategy allows for building initial training sets of higher quality than generic approaches based on uniform random sampling.

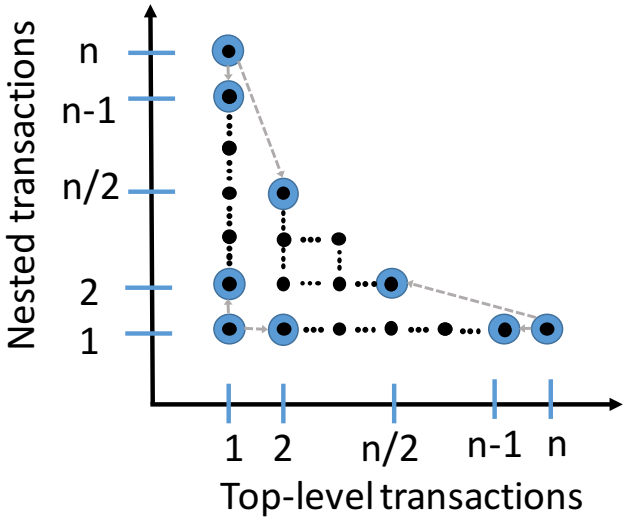


Figure 6.4: Biased sampling strategy of the initially explored configurations.

6.4.2 SMBO-driven Optimization

SMBO is a strategy for optimizing an unknown function $f : D \rightarrow \mathbb{R}$, whose estimation can only be obtained through the observation of sampled values. It operates as follows: (i) evaluate the target function f at n initial points $x_1 \ldots x_n$ and create a training set S with the resulting $\langle x_i, f(x_i) \rangle$ pairs;

(*ii*) fit a probabilistic model M over S; (*iii*) use an *acquisition function* $a(M, S) \rightarrow D$ to determine the next point x_m; *iv*) evaluate the function at x_m and accordingly update M; *v*) repeat steps (*ii*) to *iv*) until a stopping criterion is satisfied.

Acquisition function. SMBO can be coupled with different acquisition functions, and we use Expected Improvement (EI) [107].

EI selects the next point to sample based on the predicted gain with respect to the currently known optimal configuration, while keeping into account the possible uncertainty in that prediction. More formally, considering without loss of generality a maximization problem, let D_e be the set of evaluation points collected so far, D_u the set of possible points to evaluate in D and $x_{max} = \arg\max_{x \in D_e} f(x)$. Then the positive improvement function I over $f(x_{max})$ associated with sampling a point x is $I(x) = max\{f(x) - f(x_{max}), 0\}$. Since f has not been evaluated on x, $I(x)$ is not known *a priori*; however, one can use the model M, trained over past observations, to predict the expected value for the positive improvement:

$$EI(x) = \mathbb{E}[I(x)] = \int_{f(x_{max})}^{\infty} (c - f_{x_{max}}) p_M(c|x) dc$$

where $p_M(c|x)$ denotes the probability density function that the model M associates with possible outcomes of the evaluation of f at point x [107]. This formulation associates a high EI value either with points that are predicted by the model to have a high mean value or with points for which the model is uncertain about (i.e., they have high predicted variance).

By using the EI as acquisition function for SMBO, one achieves the effect of balancing exploitation and exploration: on the one hand, the model's confidence is exploited to sample the function at points that are likely to correspond to maxima; on the other hand, by exploring regions for which the model is uncertain, one provides the model with valuable information and iteratively shrink uncertainty zones.

Computing $\mathbf{p}_M(\mathbf{c}|\mathbf{x})$. Like in other previous works that applied SMBO [107], in order to ensure tractability, we assume $\mathrm{p}_M(\mathrm{c}|\mathrm{x})$ to have a Gaussian distribution $\sim N(\mu_x, \sigma_x^2)$. This allows computing EI in closed form:

$$E[I(x)] = (\mu_x - f_{max})\Phi(\frac{\mu_x - f_{max}}{\sigma_x}) + \sigma\phi(\frac{\mu_x - f_{max}}{\sigma_x}) \qquad (6.1)$$

where Φ and ϕ represent, respectively, the probability density function and cumulative distribution function of a standard Normal distribution.

In order to estimate μ_x, σ_x^2 in Eq. 6.1, AUTOPN relies on an ensemble (i.e., a set) of black-box regressors, based on the M5P algorithm [12] (described next). More in detail, the AUTOPN builds a *bagging* ensemble [107] of k M5P-based learners, each trained with a random subset (obtained via uniform sampling with replacement) of the whole training set. μ_x and σ_x^2 are computed, respectively, as the average and variance of the predictions of the ensemble of learners. We use a set of 10 bagged learners in AUTOPN, which we found to be sufficiently large to generate sufficient model diversity while incurring negligible overheads (Section 6.6.5).

Model construction. As mentioned, AUTOPN relies on M5P-based models to predict the performance of unexplored configurations. The M5P algorithm allows for constructing decision-tree based regressors that maintain on their leaves a multivariate linear model, allowing for approximating arbitrary functions by means of piece-wise linear models.

One may include a broad range of different metrics (or *features*) in the training set that is fed to the M5P regressor (e.g., abort rate, reads vs. writes ratio, etc). The larger the number of metrics, the larger the potential to provide the model with additional information and enhance its predictive power; larger feature spaces, though, require exponentially larger data sets (the, so-called, curse of dimensionality problem [107]) and accordingly larger computational costs.

In AUTOPN we address this trade-off by using a minimalistic feature space defined solely over $t \times c$, i.e., the training set fed to M5P is composed of tuples $\langle t, c \rangle \rightarrow KPI(t, c)$. This choice is based on the following rationale:

- In AUTOPN, we want to be able to use models based on the knowledge of a very small, online explored, number of configurations. Furthermore, the SMBO phase can still be corrected by the final hill-climbing-based search. These two observations allow us to use simpler models, which can be trained with fewer data and at a reduced computation cost, in the SMBO phase.

- In AUTOPN, we wish to update (re-train) the ensemble of M5P learners, and accordingly query them to obtain, in an online fashion, updated predictions of unknown configurations; ideally, this should occur whenever new samples become available.

Decision tree algorithms are known for their relatively high computational efficiency (compared, e.g., to popular regressors like ANN (Artificial Neural

Network) [108]). Yet, in order to minimize the costs related to training/-querying online the models, it is clearly desirable to use simple models.

As a consequence of this design decision, autoPN builds a different model for each workload, and it does not attempt to detect similarities with previously optimized workloads (unlike other self-tuning schemes do [7]). Besides reducing model complexity and instrumentation overheads, this choice has the key pragmatical advantage of allowing autoPN to operate fully online, which avoids the complexity and cost of gathering, offline, a training-set containing workloads representative of the target application and platform.

Stopping criterion. The EI framework provides us with a natural framework also to construct a model-driven criterion to stop exploration and settle with the best configuration encountered so far. More in detail, autoPN concludes the SMBO optimization phase as soon as the EI falls below a threshold (typical values are 1%-10%). We evaluate the effectiveness of this policy in Section 6.6.3.

Dynamic workloads. The presented design assumes that, throughout the optimization process, the application's workload does not vary, so that the KPIs of the explored configurations can be compared in a meaningful way. We argue that this assumption is realistic, since, as we will show in Section 6.6, autoPN normally requires exploring only a very small number of configurations. Indeed, this assumption holds true for all the benchmarks we considered in Section 6.6, which are representative of realistic PN-TM applications.

Yet, autoPN can easily be extended to cope with dynamically shifting workloads (again, under the assumption that the workloads vary slow enough to ensure convergence of the optimization process), by coupling it with a change detector (e.g., based on the CUSUM algorithm [109]). This would allow for identifying statistically relevant alteration of the workload characteristics (e.g., sudden throughput changes) and, accordingly, activate a new self-tuning process.

6.5 KPI Monitor and Actuator

- **KPI Monitor.** As mentioned, the key challenge addressed by the KPI monitor is to strike a trade-off between accurate and responsive measurements. In order to achieve good measurement's accuracy, the general practice in the TM self-tuning domain is to use conservative

measurements intervals, statically defined on the basis of either a fixed time period or having collected a given number of relevant events, i.e., commits of top-level transactions. None of these solutions are robust and generic, though. Policies based on monitoring windows of fixed time, e.g., [83], require a careful tuning that is strongly workload-dependent. For instance, the throughput of different TM applications can easily vary by 6 or more orders of magnitudes, from a few to millions of committed transactions per second. Using conservative values, in order to ensure good accuracy as with low-throughput workloads, can severely hinder the reactivity of the whole self-tuning process with high-throughput workloads, especially if the applications being optimized last for relatively short periods. The use of aggressive values may, conversely, lead to feeding the optimizer with erroneous information, and hinder its predictive capabilities.

Policies based on gathering a fixed number of commits, e.g., [10], conversely, are vulnerable when the system adopts a "bad" (performance-wise) configuration, e.g., where transactions starve due to excessively high contention levels, failing to commit or committing at a very slow rate. We shall illustrate the limitations of these static policies in Section 6.6.4.

In AUTOPN we address this problem by using an adaptive policy that can automatically adjust the monitoring frequency in a robust and workload independent way. This adaptive policy is based on two complementary mechanisms.

The first mechanism is based on the idea estimating the statistical uncertainty associated with the current throughput measurement on the basis of the coefficient of variation (CV). More precisely, we evaluate throughput upon each commit event since the beginning of the monitoring window (time $t = 0$). Denoting as $time(i)$ the time elapsed since the beginning of the measurement window and the occurrence of the i-th commit, the throughput upon the i-th commit, $\mathcal{T}(i)$ is simply $i/time(i)$. We then use as an estimate of the accuracy of the measurement after i commits the CV of $\mathcal{T}(i)$, i.e., $CV(\mathcal{T}(i)) = \frac{std\_dev(\{\mathcal{T}_1,...,\mathcal{T}_i\})}{avg(\{\mathcal{T}_1,...,\mathcal{T}_i\})}$. Typical CV values used in engineering to express high confidence span in the range [1%,10%]. As we will see in Section 6.6.4, 10% represents a robust value in the context of PN-TM systems.

The second mechanism is based on an adaptive time-out mechanism and is aimed at avoiding that the monitoring system gets stuck for an arbitrarily long time in a "bad" configuration. The intuition here is to use the throughput of the (1,1) configuration, corresponding to a single-threaded/sequential configuration, as a way to automatically establish a (workload-dependent) threshold that we then use to decide whether to time out the monitoring interval. Denoting with $\mathcal{T}(1,1)$ the throughput of the (1,1) configuration (which, recall, is always included in the initially sampled configurations), we can estimate the average time to experience a commit event in the (1,1) configuration as $1/\mathcal{T}(1,1)$. Waiting longer than such a time interval without witnessing any commit event suggests that the current one is likely to be a low-quality configuration — especially considering that PN-TM workloads are expected to scale, so the throughput in the (1,1) configuration is typically much lower than in the optimal configuration. Given that the quality of this configuration is very likely to be very far away from optimum, we argue that it is indeed pointless, for self-tuning purposes, to spend the further time to achieve high accuracy in its measurement. Based on these domain-specific insights, we use $1/\mathcal{T}(1,1)$ as conservative time-out value, after which we terminate the measuring interval, even if the CV-based estimator cannot confirm the measurement's stability, yet.

- **Actuator.** The actuator relies on two complementary mechanisms to dynamically adapt the parallelism degree of a PN-STM application. On the one hand, we seek to achieve total transparency for applications (e.g., legacy ones), in order to allow their optimization without requiring any modification to their source-code. This is achieved by intercepting the calls to begin and commit/abort transactions (both top-level and nested) and ensuring, via the use of semaphores, that the number of concurrent top-level transactions/nested transactions per tree is at any point in time less than allowed by the current configuration. On the other hand, we allow applications to take advantage of the knowledge on their optimal degree of inter-/intra-concurrency by exposing this information via an ad-hoc API. This information can be used, for instance, to optimize the different data partitioning schemes.

6.6 Experimental Study

In this section, we evaluate experimentally autoPN, seeking answers to the following key questions:

1. How does the optimization process employed by autoPN fare compared with the alternative, online learning approaches? (Section 6.6.2)
2. How relevant are the domain-specific optimizations integrated in the design of autoPN? In particular, how effective is the proposed biased sampling strategy for the initial configurations (Section 6.6.3) and the adaptive policy used by the KPI monitor? (Section 6.6.4)
3. What is the overhead caused by autoPN's self-tuning process and to what extent does it interfere with the performance of TM applications? (Section 6.6.5)

All the experimental data was gathered by deploying autoPN on a machine equipped with four AMD Opteron 6168 processors (48 cores total), 128GB of RAM, Linux 2.6.32 and an OpenJDK 64-bit Server 1.7.0 (build 19.0-b03) JVM.

6.6.1 Benchmarks and baseline algorithms.

In the following, we will consider 10 workloads, generated via 3 different benchmarks, which are representative of different application domains and have heterogeneous characteristics. In particular, we considered two well-known benchmarks, TPC-C and Vacation of the STAMP benchmark's suite [92], which were adopted, by previous works, to operate in a PN-STM environment [73]. We use these benchmarks to generate 3 workloads each, characterized by low, medium and high degrees of contention. We further developed a synthetic micro-benchmarks, called Array, in which we use nested transactions to parallelize the access of top-level transactions to a large, shared array of integers. We use Array to generate 4 workloads, in which transactions scan the entire array and change respectively none, 0.01%, 50% and 90% of the array's elements. It should be noted that the best average configuration over all the workloads (i.e., 24 top level and 2 nested transactions) has an average Distance From Optimum of 21.8%, its 90-th percentile is 2.56× worse than optimum and, in the worst case (Array high contention) 3.22× slower.

We consider the following five baseline algorithms: i) random search, which selects a configuration uniformly at random; ii) grid search, which explores the bi-dimensional search space by progressively sweeping first c (child transactions) and then t (top-level transactions); iii) a plain hill-climbing (HC) algorithm that starts from a randomly selected point; iv) Simulated Annealing (SA) [102], a probabilistic algorithm inspired by the thermodynamics of metals, which extends the hill-climbing algorithm by forcing it to pick a sub-optimal neighbour with a probability that decays over time, analogously to how electrons' speeds decay over time in a metal after an initial heating; v) Genetic Algorithms (GA), a family of search heuristics inspired by biological evolution in nature. GA encodes candidate solutions (i.e., configurations in our case) via bit strings, denoted as *chromosomes*. Solutions are selected via an *evolution process*, which identifies the best solutions (called elites) in the current generation, and probabilistically applies mutations (random flips of chromosomes' bits) and crossovers (swapping segments of the chromosome's between elites).

SA and, in particular, GA, require tuning a relatively large number of meta-parameters in order to be properly used: the cooling rate and initial temperature, for SA, the population size, chromosome encoding function, the rate of elitism, crossover, and mutation, among others, for GA. In order to ensure the proper configuration of these meta-parameters, we used 10-fold cross-validation combined with grid-search to compare, offline, the performance of these methods when using different settings of these meta-parameters and identify their most robust parametrization across the whole set of workloads.

For the random and grid search heuristics, we stop exploration when the last 5 explorations do not improve more than 10%. This is to provide a fair comparison with AUTOPN when using EI less than 10% as a stopping criterion.

6.6.2 Comparison with the baselines

Figure 6.5 reports the accuracy, evaluated in terms of distance from optimum (expressed in %), over time, of the baselines mentioned above, AUTOPN and a variant of AUTOPN that skips the final, hill-climbing-based local search.

In this experiment, we feed the optimizers with off-line collected traces, obtained by evaluating exhaustively every configuration in the solution space (which encompasses 198 configurations, given that we use as target system

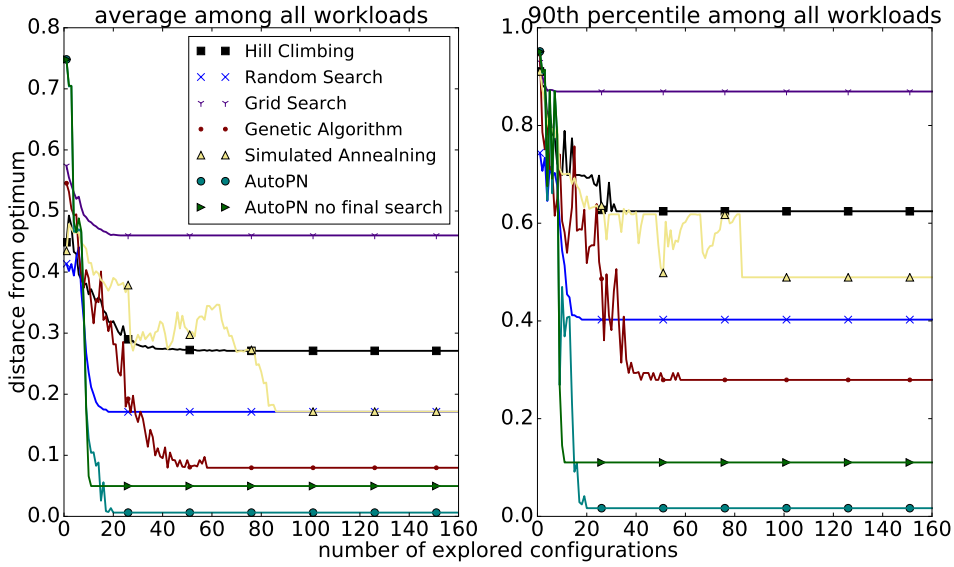


Figure 6.5: Evaluating the effectiveness of AUTOPN's optimization scheme.

a machine equipped with 48 cores). Each configuration was tested 10 times, using runs lasting 10 minutes. This choice allows us to decouple the problem of obtaining timely, but accurate, KPI measurements, which we will address in Section 6.6.4, from that of building effective optimization policies, which we can compare using fair and reproducible inputs.

The plots in Figure 6.5 report the average (left) and 90-th percentile of the distance from optimum across all the workloads. To account for the non-determinism of the optimization process, we repeat each workload 10 times for all the optimization algorithms. The plots clearly highlight that purely local search heuristics, like HC, fall easily in local optima when faced with PN-STM workloads, and are even worse than simple random search. This result is relevant, considering that hill-climbing has been used with success in the past to optimize STM systems that do not support parallel nesting. The use of random deviations from the local gradient, used by SA, only mildly ameliorates the problem. Among the considered baselines, GA is definitely the best performing one, converging eventually at around 8% from global optimum. Yet, we observe that GA is also quite disappointing in terms of convergence speed, exploring on average around 30% of the whole search space before stabilizing. We argue that, compared to HC and SA,

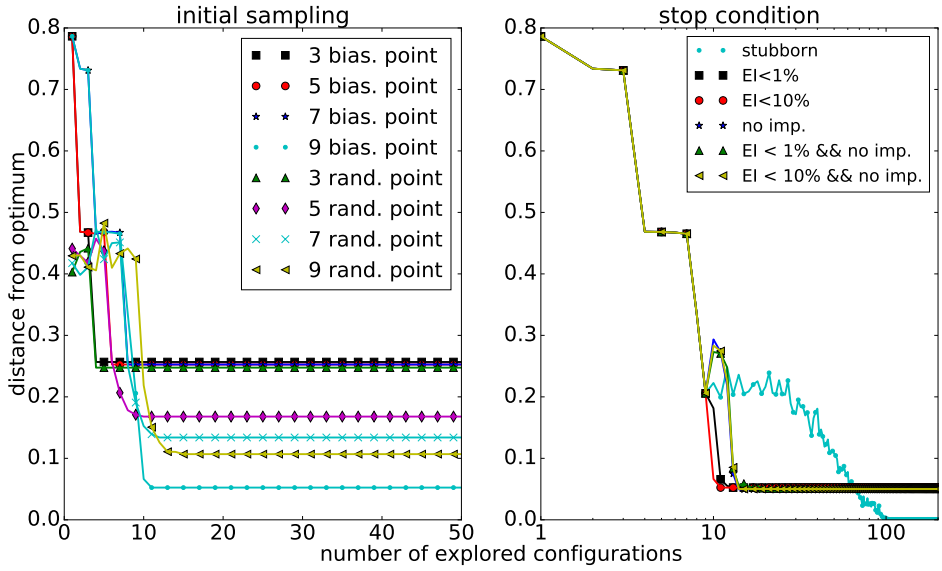


Figure 6.6: Initial sampling and stop condition in autoPN.

GA promotes a broader search in the solution space, which eventually pays off, but that is also "data greedy", probably due to its preponderant random nature.

These experimental results clearly highlight the superiority of autoPN's optimizer vs all the baseline solutions, in terms of both convergence speed and quality of the final configuration: on average, autoPN achieves 1% distance from optimum, while exploring 3× fewer configurations than the best baseline (GA).

Finally, this plot allows us to quantify the accuracy gains stemming from the use of the final refinement phase: with as few as a handful additional explorations, the final localized search reduces the distance from optimum from 5% to 1% on average, and from 10% to 2% on the 90-th percentile; a remarkable increase, in relative terms.

6.6.3 Initial sampling strategy and stop condition

We now focus on assessing the efficacy of the mechanisms employed by autoPN's optimizer to i) build the initial knowledge base for the SMBO-driven optimization, and ii) establish the completion of the SMBO process.

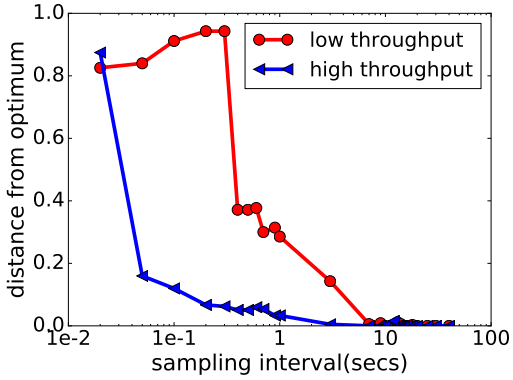
We start by comparing the accuracy achievable by the SMBO phase when using as initial sampling policy either i) 3, 5, 7 and 9 configurations selected uniformly at random, or ii) the proposed biased scheme, but selecting a smaller number of configurations, namely 3, 5, 7[1]. We disable the hill-climbing based search phase, to focus solely on the SMBO phase, and use autoPN's default stopping policy (EI<10%).

The *Initial Sampling* in Figure 6.6 shows two main trends: i) at parity of explored configurations, the biased sampling policy achieves, on average, better accuracy than a uniform random sampling scheme but only when it includes all the 9 points at the boundary of the solution space; ii) there is a major boost in accuracy when passing from 7 to 9 configurations in the biased sampling policy. Overall, these results confirm the effectiveness of autoPN's biased sampling scheme, and suggest that renouncing to sample any of the 9 boundary configurations tends to make it much less effective.

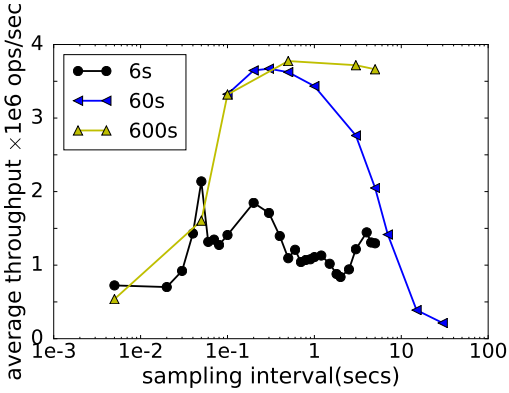
In the *Stop Condition* of Figure 6.6, we evaluate the stop condition logic employed by autoPN, when using different threshold values for the EI (1% and 10%), a heuristic (*no-improvement*) that stops exploration if the observed performance has not improved over the last K steps (we use K=5 as an illustration), hybrid heuristics resulting from the combination of EI and no-improvement, as well as an stopping condition (*stubborn*) that blocks exploration only when the optimal configuration has been found. The stubborn condition is, in fact, an ideal stop condition that cannot be implemented in practice, since the optimum is not known a priori.

The analysis of the results for stubborn reveals an interesting fact: it is much more effective to complete the SMBO phase as soon as we have identified good enough solutions, which is what EI does, than striving to achieve perfect accuracy, like stubborn does. This suggests that model-based techniques, due to their inherently approximate nature, are effective in identifying approximate solutions, but tend to blunder when they are forced to operate at a resolution that is beyond their actual reach. Local search techniques are much more efficient at this, as seen in Figure 6.5. The plot also confirms the superiority of the EI-based stopping condition versus the simpler no-improvement heuristic as well as versus the more sophisticated hybrid schemes.

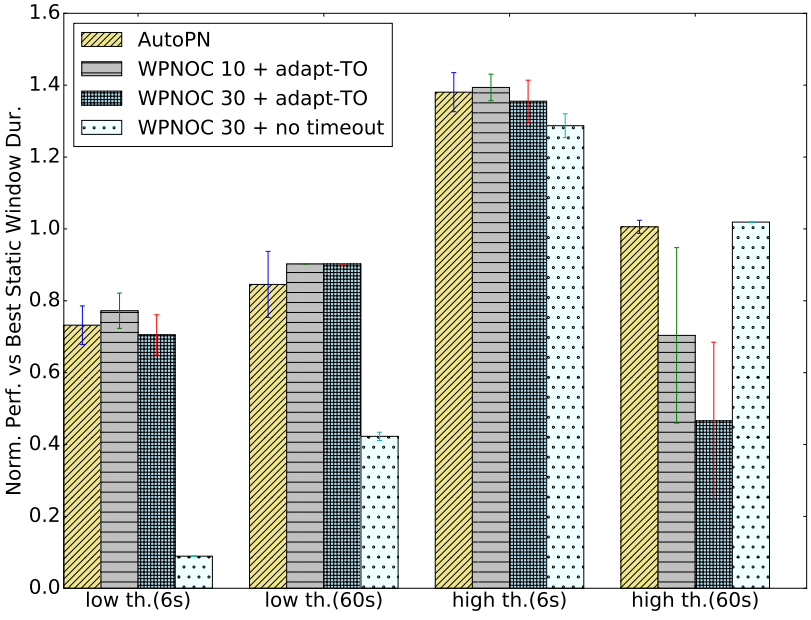
[1]When using 3 configurations, we only include the three pivots, i.e., {(1,1),(n,1),(1,n)}. With 5 configurations, we also include {(n-1,1),(1,n-1)}, and with 7 also {(2,1),(1,2)}



(a) Tuning of static time-based schemes: impact on the final solution.



(b) Tuning of static time-based schemes: impact avg. throughput.



(c) Comparing with static schemes.

Figure 6.7: Evaluating the effectiveness of AUTOPN's monitoring scheme.

6.6.4 KPI monitoring

We now move to evaluate the KPI monitoring technique used by AUTOPN. We start by considering two experiments that highlight the relevance and complexity of the problem.

In Figure 6.7a we consider a live deployment of AUTOPN, in which we execute two workloads of the Array micro benchmark, generating, respectively, low and high throughput rates. We consider a simple strategy that uses a monitoring window of a statically configured duration, which we vary on the x-axis across three orders of magnitude (from 20 msecs to 40 secs), giving AUTOPN enough time to fully complete its optimization process and report the distance from the optimum of the final solution it identified. The plot clearly shows that different workloads require different tunings: values as low as 0.1 seconds can be used for the high-throughput workloads to achieve 10% accuracy, whereas 30× larger intervals must be used to achieve similar accuracy with the other workload.

Figure 6.7b considers a more challenging, yet realistic scenario, of short-running applications: in this case, the faster is the KPI monitor in providing accurate feedback to the optimizer, the smaller the time spent exploring suboptimal configurations and the larger the average throughput for the run (reported on the y-axis). In this case, using overly conservative values can, as expectable, cripple performance severely, thus, amplifying the complexity of tuning the duration of the measurement window.

Finally, in Figure 6.7c, we contrast the performance of the adaptive monitoring policy employed in AUTOPN with: i) two variants which, instead of using the CV-based policy to detect measurement's stability, wait to track, resp. 10 and 30, commits (WPNOC10, WPNOC30), but that use AUTOPN's adaptive timeout policy (adapt-TO); iii) a policy that only waits to track 30 commits (WPNOC30). On the x-axis, we vary the workloads and their duration; on the y-axis we report the distance from the optimum of the configuration identified by AUTOPN, normalized w.r.t. to the best configuration identified using an (optimally tuned) monitoring scheme based on static measurements intervals. By the plot, we observe that AUTOPN's adaptive policy is, overall, the one to deliver the most consistent results across the considered workloads.

6.6.5 Overhead assessment

Finally, we quantify the overhead of the proposed optimization scheme. In order to isolate the self-tuning costs, we enable monitoring (using the proposed adaptive mechanism) and request the tuning algorithm to update and query its ensemble of models (based on trace-driven feedback). In the meanwhile, we inhibit the actuator from applying any configuration change. Thus, we pay the costs of self-tuning without benefiting from it. We consider an Array workload that generates no contention and scales up to all the available cores and configure the system to operate since the start in the optimal configuration. This way, we can estimate an upper bound on the overhead of autoPN's optimization scheme. Our experiments reported a negligible drop in throughput that is, on average, less than 2%, confirming the practicality of the proposed solution.

6.7 Conclusion

This chapter addresses, for the first time in the literature, the problem of optimizing the degree of inter- and intra-transaction parallelism in PN-TMs (Transactional Memory systems that support Parallel Nesting or transactional futures). We tackle this problem by proposing autoPN, an on-line self-tuning system that combines model-driven learning and localized search heuristics to achieve the best of the two approaches. We evaluate autoPN via an exhaustive experimental study, showing that autoPN reaches stability 4× faster than its counterparts, converging to solutions that are, on average, less than 1% away from optimum.

Chapter 7

Conclusions and Future Work

This dissertation has investigated how to reconcile two popular abstractions for parallel programming, namely futures and TM, by introducing the notion of *transactional future*. Chapter 3 formalized several alternative semantics for what concerns the isolation and atomicity properties of transactional futures, which explore different trade-offs between simplicity and efficiency.

The proposed semantics were implemented in two different STM systems, described in Chapter 4 and Chapter 5 and evaluated via both synthetic workloads, as well as porting of standard TM benchmarks. This allowed to quantify the performance trade-offs that arise in practical systems, when implementing the different semantics proposed in this dissertation.

Furthermore, this dissertation has filled an important gap in the literature on self-tuning of TM systems, by introducing in Chapter 6 the first mechanism capable of automatically adjusting the parallelism degree in TM systems that take advantage of intra-transaction parallelism, e.g., via the use of parallel-nesting or transactional futures.

This dissertation has opened several interesting research direction. We discuss them below, grouping them into three areas, namely:

- semantics of transactional futures,
- implementations and benchmarking of transactional futures, and
- self-tuning of TM systems that support intra-transactional parallelism.

Semantics of transactional futures. The semantics proposed in Chapter 3 of this dissertation consider as possible serialization point for a trans-

actional future either its submission or evaluation. This choice was driven by the rationale of ensuring that a transactional future is always atomically serialized either with its spawning or its evaluating transaction. We argue that this choice has the advantage of simplifying the reasoning on program's correctness, by restricting a transactional future to appear externally as if was atomically executed with the transaction that is either its logical producer or consumer.

We argue, though, that an alternative option would be allowing a transactional future to be serialized at any point in time between its submission and evaluation. An apparent advantage of such an alternative semantic is that it would be less prone to aborting. In fact, by providing more flexibility in the definition of the serialization order of a transactional future, this alternative semantic would accept a larger number of histories when compared to the semantics proposed in Chapter 3.

On the down side, this additional flexibility implies would come at a non-negligible cost for the programmer, in terms of complexity. Such an alternative semantic, in fact, would oblige programmers to reason on the correctness of their code by considering also histories in which the future executes on a snapshot produced by only some of the (sub-)transactions that compose its logical continuation.

Another research question raised by this dissertation is whether alternative methodologies could be employed for formalizing the proposed semantics for transactional futures. In this dissertation, a graph-based characterization was adopted to formalize the semantics of TM, using an approach that aligned with the notation and formalism used in classical serializability theory [18, 19]. More recently, though, alternative formalization for TM consistency criteria were proposed based on the observational refinement approach [41], which is rooted on the theory of programming languages. A natural question that arises, in the light of these considerations, is whether an equivalent formulation of the semantics proposed in this dissertation could be formalized using a similar approach based on the technique of observational refinement.

Implementations and benchmarking of transactional futures. The two STM systems presented Chapter 4 and Chapter 5 represent only two instances of the many alternative implementations of the transactional futures semantics introduced in Chapter 3. However, given the vastness of

the design space of TM systems, we believe that this dissertation has paved the way for a plethora of alternative implementations of the abstraction of transactional futures.

Besides investigating the integration of futures with other STM designs (e.g., single-versioned TM, like TinySTM), it would be interesting also to look at HTM-based implementations. Specifically, it would be interesting investigate whether the existing HTM implementations, possibly co-adjuvated by a thin software layer, e.g., [63], would be sufficient to implement the various proposed semantics for transactional futures, or whether there exist some fundamental limitations in existing HTM designs due to which it is impossible (or prohibitively) to support (some of) the proposed semantics.

Transactional futures appear to be also a perfect fit in the context of distributed TM systems [110–115], where programmers could take advantage of the asynchronous nature of execution of transactional futures to hide the latency of inter-node communication. In a distributed system settings, where the state of the TM is partitioned across multiple nodes, futures could also be used an abstraction to enhance data locality, e.g., futures spawned by a transaction executing at node n could be migrated at a different node n' if the future is likely to access a large number of data items maintained at node n', so as to ensure that those data accesses can be served locally in the same node in which the future is executed.

Another important line of future research that has been opened by this dissertation is related to the definition of standard benchmarks for transactional futures. Unfortunately, in fact, existing standard benchmarks for TMs are not fit to be parallelized using transactional futures. In particular, the key issues that we encountered while adapting existing standard benchmaks for TM systems (e.g., STAMP [92]) to transactional futures is that, with very few exceptions, the granularity of transactions in existing benchmarks is too small to benefit from futures (or any other type of intra-transaction parallelism, such as parallel nesting), as it was shown in several experiments in this dissertation. This dissertation has circumvented the lack of standard benchmarks for transactional futures by developing a number of synthetic benchmarks and by adapting the transactional logic of well-known benchmarks (Vacation [92] and TPC-C [93]). However, these efforts have only partially compensated for the current lack of standard benchmarks for transactional futures, and in the future it would be really useful for the research in this area to have access to benchmarks representative of realistic application domains appositely designed to evaluate the various trade-offs

of the proposed transactional future semantics. An example of a possible highly-relevant application domain in which the abstraction of transactional futures would find a natural fit is in JavaScript applications that rely on the asynchronous programming model [116], in which the abstraction of futures, although non-transactional ones, is already widely employed (e.g., in web-browsers, to offload tasks involving interaction with remote servers to concurrent threads executing in background).

Self-tuning of TM systems that support intra-transactional parallelism. The self-tuning approach presented in Chapter 6 of this dissertation, namely AUTOPN, determines a single value for the degree of inter- and intra-transaction parallelism of an entire application, i.e., (t, c) using the notation introduced in Section 6.1. It is easy to see that in applications that generate mixes of top-level transactions with diverse profiles (e.g., a mix of very long and very short transactions), using the same configuration of inter- and intra-transaction parallelism can lead to strongly sub-optimal performance. Given the black box nature of AUTOPN, one may argue it would be relatively straightforward to extend it to optimize N different transaction profiles by redefining its target search space as the Cartesian product N distinct (t_k, c_k) pairs (where $k \in [1, N]$), where each pair defines the ideal (inter- and intra-transactional) parallelism for a different type of transaction profile. It is unclear, though, whether the efficiency of AUTOPN would still remain acceptable when faced with a space of such a higher dimensionality.

Another limitation of AUTOPN, which would be interesting to lift in future work, is that it is designed to tune the intra-transaction parallelism degree only for sub-transactions at depth 1 (i.e., spawned directly by the top-level transaction). In order to extend AUTOPN to enable the self-tuning the intra-transaction parallelism at arbitrary deep nesting levels, one could used a methodology similar in spirit to the one described just above: increasing the dimensionality of the search space by considering tuples of the form $(t, c_1, c_2, \ldots, c_{max})$ where c_i $(i \in [1, max])$ represents the intra-transaction parallelism at depth i.

Another research line suggested by our work is how to incorporate information on the noisiness of sampled data (e.g., measured in terms of coefficient of variation) in the modeling phase. This is in contrast with the current AUTOPN's approach, in which data is fed to the model only after having ensured that the corresponding measurement is statistically meaningful (or

irrelevant for the optimization's purposes).

Bibliography

[1] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[2] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[3] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[4] Robert H Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[5] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, August 1977.

[6] Daniel Friedman and David Wise. The impact of applicative programming on multiprocessing. pages 263–272, 1976.

[7] Diego Rughetti, Paolo Romano, Francesco Quaglia, and Bruno Ciciani. Automatic tuning of the parallelism degree in hardware transactional memory. In *Euro-Par*, pages 475–486, 2014.

[8] Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia. Analytical/ml mixed approach for concurrency regulation in software transactional memory. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 81–91. IEEE, 2014.

[9] Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia. Machine learning-based self-adjusting concurrency in software transactional memory systems. In *MASCOTS*, pages 278–285. IEEE, 2012.

[10] Diego Didona, Pascal Felber, Derin Harmanci, Paolo Romano, and Joerg Schenker. Identifying the optimal level of parallelism in transactional memory applications. *Computing*, 97(9):939–959, 2015.

[11] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Adaptive concurrency control for transactional memory. In *First Workshop on Programmability Issues for Multi-Core Computers*, pages 64–71, 2008.

[12] Yong Wang and Ian H Witten. Induction of model trees for predicting continuous classes. 1996.

[13] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[14] Oracle and/or its affiliates. Javadoc of the future interface in java. In *https://docs.oracle.com/javase/8/docs/*, 2019.

[15] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for java. In *ACM SIGPLAN Notices*, volume 40, pages 439–453. ACM, 2005.

[16] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. *ACM SIGOPS Operating Systems Review*, 32(5):58–69, 1998.

[17] Alex Kogan and Maurice Herlihy. The future (s) of shared data structures. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 30–39, 2014.

[18] Christos H Papadimitriou. Serializability of concurrent database updates. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1979.

[19] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, (3):203–216, 1979.

[20] Edsger W Dijkstra. Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*, pages 289–294. Springer, 2001.

[21] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (2):125–143, 1977.

[22] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[23] Alex A Aravind. Yet another simple solution for the concurrent programming control problem. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):1056–1063, 2010.

[24] Alex Ho, Steven Smith, and Steven Hand. On deadlock, livelock, and forward progress. Technical report, University of Cambridge, Computer Laboratory, 2005.

[25] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[26] Pierre-Jacques Courtois, Frans Heymans, and David Lorge Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971.

[27] Jonathan Corbet. Big reader locks.

[28] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[29] Mathieu Desnoyers, Paul E McKenney, Alan S Stern, Michel R Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2011.

[30] Jonathan Corbet. Big reader locks, 2010.

[31] Shady Issa, Paolo Romano, and Tiago Lopes. Speculative read write locks. In *Proceedings of the 19th International Middleware Conference*, pages 214–226. ACM, 2018.

[32] Diego Didona, Nuno Diegues, Anne-Marie Kermarrec, Rachid Guerraoui, Ricardo Neves, and Paolo Romano. Proteustm: Abstraction meets performance in transactional memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 757–771. ACM, 2016.

[33] Yossi Lev, Mark Moir, and Dan Nussbaum. Phtm: Phased transactional memory. In *Workshop on Transactional Computing (Transact)*, 2007.

[34] Richard M Yoo, Christopher J Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–11. IEEE, 2013.

[35] Sérgio Miguel Fernandes and Joao Cachopo. Lock-free and scalable multi-version software transactional memory. In *ACM SIGPLAN Notices*, volume 46, pages 179–188. ACM, 2011.

[36] Vincent Gramoli and Rachid Guerraoui. Democratizing transactional programming. In *Middleware 2011*, pages 1–19. Springer, 2011.

[37] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 3–14. ACM, 2014.

[38] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[39] Rachid Guerraoui and Michal Kapalka. Opacity: A correctness condition for transactional memory. Technical report, 2007.

[40] 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. ACM, 2008.

[41] Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. Characterizing transactional memory consistency conditions using observational refinement. *Journal of the ACM (JACM)*, 65(1):2, 2018.

[42] Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. A programming language perspective on transactional memory consistency. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 309–318. ACM, 2013.

[43] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[44] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pages 522–529. IEEE, 2003.

[45] Victor Bushkov and Rachid Guerraoui. Liveness in transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, pages 32–49. Springer, 2015.

[46] Rachid Guerraoui and Michal Kapalka. The semantics of progress in lock-based transactional memory. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 404–415, 2009.

[47] Rachid Guerraoui, Thomas A Henzinger, and Vasu Singh. Permissiveness in transactional memories. In *International Symposium on Distributed Computing*, pages 305–319. Springer, 2008.

[48] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 16–25, 2010.

[49] Idit Keidar and Dmitri Perelman. On avoiding spare aborts in transactional memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 59–68, 2009.

[50] Michael F Spear, Virendra J Marathe, William N Scherer, and Michael L Scott. Conflict detection and validation strategies for software transactional memory. In *International Symposium on Distributed Computing*, pages 179–193. Springer, 2006.

[51] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *International Symposium on Distributed Computing*, pages 194–208. Springer, 2006.

[52] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

[53] Luke Dalessandro, Michael F Spear, and Michael L Scott. Norec: streamlining stm by abolishing ownership records. In *ACM Sigplan Notices*, volume 45, pages 67–78. ACM, 2010.

[54] 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. ACM, 2008.

[55] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *International Symposium on Distributed Computing*, pages 284–298. Springer, 2006.

[56] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of blue gene/q hardware support for transactional memories. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 127–136. IEEE, 2012.

[57] Harold W Cain, Maged M Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 225–236. ACM, 2013.

[58] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for ibm system z. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 25–36. IEEE, 2012.

[59] Intel Corporation. Architecture instruction set extensions programming reference. chapter 8: Intel transactional synchronization extensions. 2012.

[60] Nuno Diegues and Paolo Romano. Self-tuning intel restricted transactional memory. *Parallel Computing*, 50:25–52, 2015.

[61] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M Michael, and Hisanobu Tomari. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 144–157. IEEE, 2015.

[62] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[63] Shady Issa, Pascal Felber, Alexander Matveev, and Paolo Romano. Extending hardware transactional memory capacity via rollback-only transactions and suspend/resume. *Distributed Computing*, pages 1–22, 2019.

[64] Ricardo Filipe, Shady Issa, Paolo Romano, and João Barreto. Stretching the capacity of hardware transactional memory in ibm power architectures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 107–119, 2019.

[65] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[66] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 53–64, 2011.

[67] Pascal Felber, Shady Issa, Alexander Matveev, and Paolo Romano. Hardware read-write lock elision. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–15, 2016.

[68] Kunal Agrawal, Jeremy T Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 163–174. ACM, 2008.

[69] Joao Barreto, Aleksandar Dragojević, Paulo Ferreira, Rachid Guerraoui, and Michal Kapalka. Leveraging parallel nesting in transactional memory. In *ACM Sigplan Notices*, volume 45, pages 91–100. ACM, 2010.

[70] Haris Volos, Adam Welc, Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, Xinmin Tian, and Ravi Narayanaswamy. Nepaltm: design and implementation of nested parallelism for transactional memory systems. In *ECOOP 2009–Object-Oriented Programming*, pages 123–147. Springer, 2009.

[71] Woongki Baek and Christos Kozyrakis. NesTM: Implementing and Evaluating Nested Parallelism in Software Transactional Memory. In *Proceedings of the 9th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[72] Ranjeet Kumar and Krishnamurthy Vidyasankar. Hparstm: A hierarchy-based stm protocol for supporting nested parallelism. In *the 6th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'11)*, 2011.

[73] Nuno Diegues and Joao Cachopo. Practical parallel nesting for software transactional memory. In *Distributed Computing*, pages 149–163. Springer, 2013.

[74] Naweiluo Zhou, Gwenaël Delaval, Bogdan Robu, Éric Rutten, and Jean-François Méhaut. An autonomic-computing approach on mapping threads to multi-cores for software transactional memory. *Concurrency and Computation: Practice and Experience*, 30(18):e4506, 2018.

[75] Márcio Castro, Luís Fabrício W Góes, and Jean-François Méhaut. Adaptive thread mapping strategies for transactional memory applications. *Journal of Parallel and Distributed Computing*, 74(9):2845–2859, 2014.

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

[77] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 188–197. ACM, 2014.

[78] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[79] J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. Collaborative filtering recommender systems. In *The adaptive web*, pages 291–324. Springer, 2007.

[80] Nuno Diegues and Paolo Romano. Self-tuning intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 209–219, 2014.

[81] 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.

[82] Daniel Castro, Paolo Romano, Diego Didona, and Willy Zwaenepoel. An analytical model of hardware transactional memory. In *Proceedings of the 25th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, number EPFL-CONF-229444, 2017.

[83] Kaushik Ravichandran and Santosh Pande. F2c2-stm: Flux-based feedback-driven concurrency control for stms. In *Parallel and Dis-*

tributed Processing Symposium, 2014 IEEE 28th International, pages 927–938. IEEE, 2014.

[84] Jingna Zeng, Joao Barreto, Seif Haridi, Luís Rodrigues, and Paolo Romano. The future (s) of transactional memory. In *Parallel Processing (ICPP), 2016 45th International Conference on*, pages 442–451. IEEE, 2016.

[85] Jingna Zeng, Paolo Romano, Luís Rodrigues, Seif Haridi, and João Bareto. In search of semantic models for reconciling futures and transactional memory. 7th Workshop on the Theory of Transactional Memory, 2015.

[86] Jingna Zeng, Seif Haridi, Shady Issa, Paolo Romano, and Luis Rodrigues. Giving future (s) to transactional memory. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 587–589, 2020.

[87] Jingna Zeng, Shady Alaaeldin Issa, Seif Haridi, Luis Rodrigues, and Paolo Romano. Investigating the semantics of futures in transactional memory systems. Technical report, INESC-ID, Lisbon, Portugal, 2019.

[88] Damien Imbs and Michel Raynal. Virtual world consistency: A condition for stm systems (with a versatile protocol with invisible read operations). *Theor. Comput. Sci.*, 444:113–127, July 2012.

[89] Ivo Anjo and João Cachopo. Improving continuation-powered method-level speculation for jvm applications. In *Algorithms and Architectures for Parallel Processing*, pages 153–165. Springer, 2013.

[90] The Apache Software Foundation. The javaflow component, 2004-2008.

[91] Virendra J Marathe and Michael L Scott. A qualitative survey of modern software transactional memory systems. *University of Rochester Computer Science Dept., Tech. Rep*, 2004.

[92] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. IEEE, 2008.

[93] TPC. Tpc-c an on-line transaction processing benchmark, 2001-2015.

[94] Vincent Gramoli, Derin Harmanci, and Pascal Felber. On the input acceptance of transactional memory. *Parallel Processing Letters*, 20(01):31–50, 2010.

[95] Daniel Castro, Paolo Romano, and João Barreto. Hardware transactional memory meets memory persistency. *Journal of Parallel and Distributed Computing*, 130:63–79, 2019.

[96] Wilson WL Fung, Inderpreet Singh, Andrew Brownsword, and Tor M Aamodt. Hardware transactional memory for gpu architectures. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 296–307. IEEE, 2011.

[97] Torval Riegel, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. From causal to z-linearizable transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 340–341, 2007.

[98] Konrad Siek and Paweł T Wojciechowski. Atomic rmi: A distributed transactional memory framework. *International Journal of Parallel Programming*, 44(3):598–619, 2016.

[99] Ricardo Filipe and Joao Barreto. Nested parallelism in transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, pages 192–209. Springer, 2015.

[100] Jingna Zeng, Paolo Romano, Joao Barreto, Luís Rodrigues, and Seif Haridi. Online tuning of parallelism degree in parallel nesting transactional memory. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 474–483. IEEE, 2018.

[101] J Eliot B Moss and Antony L Hosking. Nested transactional memory: model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, 2006.

[102] Chii-Ruey Hwang. Simulated annealing: theory and applications. *Acta Applicandae Mathematicae*, 12(1):108–111, 1988.

[103] Kim-Fung Man, Kit-Sang Tang, and Sam Kwong. Genetic algorithms: concepts and applications. *IEEE transactions on Industrial Electronics*, 43(5):519–534, 1996.

[104] John Eliot Blakeslee Moss. Nested transactions: An approach to reliable distributed computing. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1981.

[105] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: a benchmark for software transactional memory. Technical report, 2006.

[106] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[107] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[108] Tom Michael Mitchell. *Machine Learning*. McGraw-Hill, 1st edition, 1997.

[109] Michèle Basseville, Igor V Nikiforov, et al. *Detection of abrupt changes: theory and application*. Prentice Hall Englewood Cliffs, 1993.

[110] Paolo Romano, Luis Rodrigues, Nuno Carvalho, and Joäo Cachopo. Cloud-tm: harnessing the cloud with distributed transactional memories. *ACM SIGOPS Operating Systems Review*, 44(2):1–6, 2010.

[111] Maria Couceiro, Pedro Ruivo, Paolo Romano, and Luis Rodrigues. Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation. *IEEE Transactions on Parallel and Distributed Systems*, 26(11):2942–2955, 2014.

[112] Alexandru Turcu, Binoy Ravindran, and Roberto Palmieri. Hyflow2: A high performance distributed transactional memory framework in scala. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 79–88, 2013.

[113] Sachin Hirve, Roberto Palmieri, and Binoy Ravindran. Archie: a speculative replicated transactional system. In *Proceedings of the 15th international Middleware Conference*, pages 265–276, 2014.

[114] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 456–475. Springer, 2012.

[115] Sachin Hirve, Roberto Palmieri, and Binoy Ravindran. Hipertm: High performance, fault-tolerant transactional memory. *Theoretical Computer Science*, 688:86–102, 2017.

[116] Daniel Parker. *JavaScript with Promises: Managing Asynchronous Code*. " O'Reilly Media, Inc.", 2015.

TRITA-EECS-AVL-2020:50
ISBN 978-91-7873-654-6