# Scalable Synchronization in Shared-Memory Systems:

# Extrapolating, Adapting, Tuning

THÈSE N$^{O}$ 8843 (2018)

PRÉSENTÉE LE 12 OCTOBRE 2018

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE CALCUL DISTRIBUÉ

PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Georgios CHATZOPOULOS

acceptée sur proposition du jury:

Dr R. Boulic, président du jury
Prof. R. Guerraoui, directeur de thèse
Prof. P. Felber, rapporteur
Prof. V. Quéma, rapporteur
Prof. A. Ailamaki, rapporteuse

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2018

"*A journey of a thousand miles*
*begins with one step*"
— Laozi

To my parents, Dimitris and Lemonia,
my sister Fenia, and my brother Evangelos

# Acknowledgements

First, I would like to thank my advisor, Prof. Rachid Guerraoui. Rachid taught me a lot, both at an academic and a personal level, including what good research is and how it should be done, as well as how to think about problems. Without his support and advice, this journey would not have been possible.

I would like to also thank Aleksandar Dragojević, whom I have worked with for the biggest part of this PhD, for his help, both at a technical and a personal level, his support and patience, as well as all the nice times we had, both during my internship at Microsoft Research and every time we would meet.

I want to thank Prof. Anastasia Ailamaki, Prof. Vivien Quéma and Prof. Pascal Felber for their feedback on this dissertation, and Dr. Ronan Boulic for presiding over my thesis committee.

Many thanks to all the members of the Distributed Computing Laboratory: Oana Balmau, Peva Blanchard, Victor Bushkov, Mahdi El Mhamdi, Mihai Letia, Alexandre Maurer, Rhicheek Patra, Julien Stainer, Mahsa Taziki, Kristine Verhamme, Jingjing Wang and Igor Zablotchi. A big thanks to Lê Nguyên Hoang for helping with the translation of my abstract to French. During my PhD, I was also lucky enough to meet some special people at DCL, whom I consider close friends. Adrian, David, George, Karolos, Matej, Tudor, and Vasilis made the days and nights of the PhD much more enjoyable and created an environment in which I really felt at home, and for that I am deeply grateful. I would like to thank Vasilis for also being a great collaborator, and for contributing to the research in this thesis. A big thanks to France Faille for making my life significantly easier by helping with everything related to (among other things) administrative tasks, as well as for all the interesting discussions in French. Finally, thanks to the lab's system administrator, Fabien Salvi, for making everything technically possible.

Many people outside the lab were also very important these five years. I would like to thank George Psaropoulos, whom I also shared apartments with when I first arrived in Switzerland, as well as Danica, Eleni, Iraklis, Javier, Manos and Matt for their company and help, both personally and professionally. I would also like to thank all of my friends in Greece and abroad. There are too many of them to thank and too much to write for each one, but they all know who they are and how much they have done for me, being there for me for many years now, and enduring my unpredictable schedule and last-minute notices (and cancellations). A big "thank you" goes to William Markoutis, who has been there from the beginning, and has honoured me with his friendship and kind character for more than 20 years.

**Acknowledgements**

These past five years I had the luck of meeting some great people in the ShARE student association that I have been part of, including members at EPFL, as well as members from all over the world. I would like to thank everyone that I had the chance to meet and work with.

A very special "thank you" goes to Mirjana Pavlović, for her love and support, for putting up with my never-ending supply of "ideas" and busy schedule, and for always encouraging me to do more, both in the office and outside. She made my life much more colourful and sunny.

Last, but in no way least, I would like to thank my family, my parents Dimitris and Lemonia, my sister Fenia, and my brother Evangelos, for shaping the person I am today and for supporting me unconditionally in my every step. They have always been the most important "constants" in my life and for that I can never thank them enough.

*The research presented in this dissertation has been supported by the European Research Council (ERC) Grant 339539 (AOC) and the "Revisiting Transactional Computing on Modern Hardware" project of the Microsoft-EPFL Joint Research Center.*

*Lausanne, 21 September 2018* Georgios Chatzopoulos

# Preface

The research presented in this dissertation was conducted in the Distributed Computing Laboratory at EPFL, under the supervision of Professor Rachid Guerraoui, between 2013 and 2018. The main results of this dissertation appear originally in the following publications:

1. G. Chatzopoulos, A. Dragojević and R. Guerraoui. "*ESTIMA: Extrapolating ScalabiliTy of In-Memory Applications*". In the Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16), 2016 [29].

2. J. Antić, G. Chatzopoulos, R. Guerraoui, and V. Trigonakis. "*Locking Made Easy*". In the Proceedings of the 17th International Middleware Conference (Middleware '16), 2016 (Best paper award) [14].

3. G. Chatzopoulos, A. Dragojević and R. Guerraoui. "*ESTIMA: Extrapolating ScalabiliTy of In-Memory Applications*". In ACM Transactions on Parallel Computing (TOPC) - Special Issue: Invited papers from PPoPP 2016, Part 2, 2017 [30].

4. G. Chatzopoulos, A. Dragojević and R. Guerraoui. "*SPADE: Tuning scale-out OLTP on modern RDMA clusters*". To appear in the Proceedings of the 19th International Middleware Conference (Middleware '18), 2018.

Besides the above-mentioned publications, I was also involved in other research projects that resulted in the following publication:

1. G. Chatzopoulos, R. Guerraoui, T. Harris, and V. Trigonakis. "*Abstracting Multi-Core Topologies with MCTOP*". In the Proceedings of the 12th European Conference on Computer Systems (EuroSys '17), 2017 [31].

# Abstract

As hardware evolves, so do the needs of applications. To increase the performance of an application, there exist two well-known approaches. These are scaling up an application, using a larger multi-core platform, or scaling out, by distributing work to multiple machines. Both approaches are typically implemented using the *shared-memory* and *message-passing* programming models. In both programming models, and for a wide range of workloads, there is contention for access to data, and applications need *synchronization*. On modern multi-core platforms with tens of cores and network interconnects with sub-millisecond latencies, synchronization bottlenecks can significantly limit the scalability of an application.

This dissertation studies the scalability of synchronization in shared-memory settings and focuses on the changes brought upon by modern hardware advances in processors and network interconnects. As we show, current approaches to understanding the scalability of applications, as well as prevalent synchronization primitives, are not designed for modern workloads and hardware platforms. We then propose methods and techniques that take advantage of hardware and workload knowledge to better serve application needs.

We first look into better understanding the scalability of applications, using low-level performance metrics in both hardware and software. We introduce ESTIMA, an easy-to-use tool for extrapolating the scalability of in-memory applications. The key idea underlying ESTIMA is the use of stalled cycles in both hardware (e.g., cycles spent waiting for cache line fetches or busy locks), as well as software (e.g., cycles spent on synchronization, or on failed Software Transactional Memory transactions). ESTIMA measures stalled cycles on a few cores and extrapolates them to more cores, estimating the amount of *waiting* in the system.

We then focus on locking, a common way of synchronizing data accesses in a shared-memory setting. Typically, contention for data limits the scalability of an application. A poor choice of locking algorithm can further impact scalability. We introduce GLS, a middleware that makes lock-based programming simple and effective. In contrast to classic lock libraries, GLS does not require any effort from the programmer for allocating and initializing locks, nor for selecting the appropriate locking strategy. GLS relies on GLK, a generic lock algorithm that dynamically adapts to the contention level on the lock object, delivering the best performance among simple spinlocks, scalable queue-based locks, and blocking locks.

Finally, we study the performance of distributed systems that offer *transactions*, by exposing shared-memory semantics. Recently, such systems have seen renewed interest due to

advancements in networking hardware for datacenters. We show how such systems require significant manual tuning to achieve good performance in a popular category of workloads (OLTP), and argue that doing so is error-prone, as well as time-consuming, and it needs to be repeated when the workload or the hardware change. We then introduce SPADE, a physical design tuner for OLTP workloads on modern RDMA clusters. SPADE automatically decides on data partitioning, index and storage parameters, and the right mix of direct remote data accesses and function shipping to maximize performance. To achieve this, SPADE uses low-level hardware and network performance characteristics gathered through micro-benchmarks.

**Keywords:** Multi-cores, scalability, synchronization, extrapolation, performance counters, locking, adaptive locks, RDMA, tuning, OLTP

# Résumé

Les besoins des applications évoluent avec les progrès en hardware. Il existe deux façons bien connues pour améliorer les performances d'une application : utiliser une plateforme multi-cœur, ou distribuer le calcul sur de multiples machines. Ces deux approches sont typiquement implémentées à l'aide de modèles de *mémoire partagée* et de *passages de messages*. Pour une grande gamme de charges de calculs, il y a d'étranglement au niveau de l'accès aux données, et les applications ont besoin de *synchronisation*. Sur les plateformes modernes avec des dizaines de cœurs et des interconnections de réseaux performantes, les exigences de synchronisation peuvent significativement limiter la mise à l'échelle d'une application.

Cette dissertation étudie la mise à l'échelle de la synchronisation sur la mémoire partagée. Elle s'intéresse aux changements exigés par les avancées modernes en hardware des processeurs et des réseaux. Comme on le montre, les approches traditionnelles pour comprendre la mise à l'échelle de la synchronisation, ainsi que les primitives de synchronisation prédominantes, ne sont pas adaptées aux charges et aux plateformes hardware modernes. On propose des méthodes et des techniques qui exploitent au mieux la connaissance de la charge et du hardware pour répondre aux besoins des applications.

On commence par chercher à mieux comprendre la mise à l'échelle des applications, à l'aide de métriques de performance de bas niveau, à la fois en termes hardware et software. On introduit ESTIMA, un outil pour extrapoler la scalabilité d'applications. L'idée clé derrière ESTIMA est l'utilisation de cycles retardés, en hardware (e.g., les cycles des requêtes de caches perdus) aussi bien qu'en software (e.g., les cycles passés en synchronisation, ou les transactions en *Software Transactional Memory* qui ont échoué). ESTIMA mesure les cycles retardés dans quelques cœurs et les extrapole, en estimant le temps que le système passe à attendre.

On se concentre ensuite sur le verrouillage, une approche commune de synchronisation. La contention pour l'accès aux données typiquement limite la mise à l'échelle d'une application. Un mauvais choix d'algorithme de verrouillage peut davantage impacter sa scalabilité. On introduit GLS, un middleware qui rend la programmation avec des verrous simple et efficace. Contrairement aux bibliothèques classiques, GLS ne requiert aucun effort de la part du programmeur pour allouer et initialiser les verrous, ni pour choisir l' algorithme. GLS utilise GLK, un algorithme de verrouillage générique qui adapte dynamiquement au niveau de contention pour chaque verrou, atteignant la meilleure performance parmi des verrous tournants simples, des verrous scalables à base de queue, et les verrous de blocage.

Enfin, on étudie les systèmes distribués qui permettent les *transactions*. On montre que de tels systèmes nécessitent d'importants réglages à la main pour atteindre de bonnes performances pour des charges OLTP. On se soutien que cette approche est chronophage et sujette aux erreurs. On introduit SPADE, un système de réglage physique pour les charges OLTP sur les clusters RDMA. SPADE décide automatiquement du partitionnement des données, des paramètres de stockage et d'indexage, et du bon mix entre accès directs et le lancement de fonction à distance pour maximiser les performances. Pour y arriver, SPADE utilise les caractéristiques de performance de hardware et de réseau, collectées via des micro-benchmarks.

**Keywords :** Multi-cœurs, scalabilité, synchronisation, extrapolation, compteurs de performance, verrouillage, verrou adaptatif, RDMA, réglage, OLTP

# Contents

# List of Figures

# List of Tables

# 1 Introduction

In 1965, Gordon Moore described a doubling of the density of components in integrated circuits every two years. This observation is nowadays known as *Moore's Law* [118]. For decades, Moore's Law went hand in hand with the Dennard Scaling Rule, which described that as circuit density increases, power density remains constant, resulting in an increase of the performance of circuits [42]. The result was increased performance for applications without significant effort from the part of the developers, as each generation of processors would bring performance improvements.

In the early 2000s, the Dennard Scaling Rule's validity came to an abrupt end, when current leakage prevented further lowering of the threshold voltages, limiting transistor performance gains [23]. Processor designers moved towards utilizing the increasing transistor density to add more processing cores on the same die. Commodity processors became *multi-core processors*. Applications stopped seeing performance improvements and developers began transforming their applications into multi-threaded ones. As modern processors keep adding more cores and features such as deep cache hierarchies, multi-threaded applications take advantage of hardware, scaling their performance with the resources available in a machine. We say that such applications *scale up*. Additionally, even before processors hit the end of Dennard scaling, applications that needed more resources than a single machine could offer, opted for distribution of work. That refers to systems consisting of multiple machines, often interconnected by fast networks (as allowed by each generation of networking hardware), and applications that spread work among these machines. We say that such applications *scale out*.

Developing applications for both scaling methods involves distributing work between the processes that make up the application. These processes might run on the same machine, or different machines. For some workloads, called *embarrassingly parallel*, distributing the work is straightforward, and minimal communication is necessary between the processes. Such applications typically scale with very little effort, as the resources available (processing cores and/or machines) increase. However, for a wide range of applications, distributing work is not as trivial, and processes have *shared state*. In those cases, to ensure correctness, there is a need for *synchronization*, that is coordination of access to this shared state.

Applications nowadays follow one of two main programming models: *shared-memory*, or *message-passing*. In the first model, all the threads of an application have access to a common memory, and communicate by accessing common variables, or data structures. Examples of such communication are shared counters, shared flags, and more. In the second model, threads compute independently, and communicate with other threads when there is a need for exchange of state. For example, threads can compute partial results, exchange their computed parts, and continue computing towards a final result.

While the two programming models have typically lent themselves to the two scaling methods, with shared memory being primarily used for scaling up and message-passing used to scale out, the boundaries have become blurred, as modern hardware has evolved. Modern multi-core systems resemble small-scale distributed processing cores, and research projects have proposed message-passing inside a single machine [19]. Similarly, shared-memory abstractions can be built on top of a cluster of machines, where the global address space can be distributed over the physical memory of the machines. Such implementations have typically been prohibitively expensive computationally. However, modern network interconnects have offered fast ways to access remote memory, through Remote Direct Memory Access capabilities. This advancement has made it feasible to offer shared-memory abstractions over clusters of interconnected machines, using commodity hardware [8, 32, 52, 88, 106, 174].

In this dissertation, we focus on shared-memory model systems and applications. In such systems, memory is exposed to all the threads of an application, in a shared global address space. Threads are free to read and write any part of the memory. Memory can either be a physical memory module attached to a processor running all the threads of an application, a physical memory module attached to a remote processor on the same machine (i.e., a different Non-Uniform Memory Access Domain), or physical memory attached to a remote machine exposed over a local network. In such distributed settings, we only focus on machines interconnected in a local environment with fast network interconnects, such as a datacenter.

## 1.1 Shared-Memory Scalability and Synchronization

Synchronization on shared-memory systems is traditionally achieved through *locking*. Developers choose the granularity of data that need to be protected from concurrent accesses (e.g., a node in a linked list) and choose a locking algorithm to implement their locking strategy. During execution, the algorithm acquires and releases locks, based on the operation to be performed. Synchronization through locking can be a significant bottleneck, but can also affect the correctness of an application, since lock implementations can suffer from problems such as deadlocks and livelocks [33, 76]. Another method of synchronizing access to data is through *lock-free* algorithms [63, 77], which typically use hardware-provided primitives (such as *Compare-and-Swap*) to avoid using locks.

Alternatively, applications can use *transactions*. Transactions are indivisible sets of operations in an application, that typically satisfy a series of properties. In shared-memory systems,

applications that use transactions rely on either hardware-provided abstractions (e.g., Intel's TSX [81]), or runtimes that offer transactional abstractions (e.g., SwissTM [49]).

Nowadays, modern multi-core machines become bigger and more diverse. Processors on server machines typically contain tens of cores, and hundreds of gigabytes of main memory. This allows them to achieve performance by holding big parts (or the entirety) of their dataset in main memory. At the same time, it also means that new bottlenecks are becoming more important. Synchronization is one of them. The performance degradation of synchronization algorithms and implementations that are not designed to scale to this large number of cores is significant. If the synchronization primitives in a synchronization-heavy application do not scale, the scalability of the application suffers.

In this dissertation, we focus on understanding and improving the scalability of synchronization in the shared-memory setting. We propose techniques and methodologies for predicting the scalability potential of applications and understanding the bottlenecks that may limit scalability. We then set out to improve two common synchronization primitives used in shared-memory applications: locking in multi-core systems and transactions in distributed shared-memory systems. We do so, by adapting to the contention and workload, showing that static approaches can significantly limit scalability.

## 1.2 Understanding the Scalability of Shared-Memory Applications

Modern multi-core machines nowadays have large main memories, allowing for a wide range of applications to keep their data in-memory, avoiding slow secondary storage and networks. However, many application still exhibit poor performance in large production machines. Understanding the reasons behind this can be difficult: developers typically use smaller machines to develop and test applications than the ones used in production settings. Additionally, processors and memory systems evolve, meaning that applications tested today even on high-end multi-cores, in a few years will be run on significantly larger machines.

To better understand the scalability of applications, various approaches and methodologies have been proposed, including performance evaluation and detailed models of applications [70, 109]. In the first part of this dissertation, we argue that such approaches are not only time-consuming and error-prone, and look for an alternative that not only makes the process easier, but also provides high accuracy.

In Chapter 4 we present ESTIMA, an easy-to-use tool for extrapolating the scalability of in-memory applications. ESTIMA is designed to perform a simple, yet important task: given the performance of an application on a small machine with a handful of cores, ESTIMA extrapolates its scalability to a larger machine with more cores, while requiring minimum input from the user. The key idea underlying ESTIMA is the use of stalled cycles in hardware (e.g., cycles that the processor spends waiting for missed cache line fetches or busy locks) and software (focusing on cycles spent on synchronization). ESTIMA measures stalled cycles on a few cores

and extrapolates them to more cores, estimating the amount of *waiting* in the system. ESTIMA can be effectively used to predict the scalability of in-memory applications for bigger execution machines. We extensively evaluate our solution and show the effectiveness of ESTIMA on a large number of in-memory benchmarks.

## 1.3 Improving the Scalability of Shared-Memory Applications

In the second part of this thesis, we focus on two synchronization primitives commonly used in shared-memory applications: locking and transactions. We first identify the shortcomings of existing designs and implementation, and present two systems that improve the performance of these primitives, tailoring them to modern hardware platforms.

A priori, locking seems easy: To protect shared data from concurrent accesses, it is sufficient to lock before accessing the data and unlock after. Nevertheless, making locking efficient requires fine-tuning (a) the granularity of locks and (b) the locking strategy for each lock and possibly each workload. As a result, locking can become very complicated to design and debug.

In Chapter 5 we present GLS, a middleware that makes lock-based programming simple and effective. GLS offers the classic lock-unlock interface. However, in contrast to classic lock libraries, it does not require any effort from the programmer for allocating and initializing locks, nor for selecting the appropriate locking strategy. GLS is based on GLK, a generic lock algorithm that dynamically adapts to the contention level on the lock object. GLK delivers the best performance among simple spinlocks, scalable queue-based locks, and blocking locks. Furthermore, GLS offers several debugging options for easily detecting various lock-related issues, such as deadlocks. We evaluate GLS and GLK on two modern multi-core platforms, using several software systems and show how GLK improves their performance by 23% on average, compared to their default locking strategies. We also illustrate the simplicity of using GLS and its debugging facilities by rewriting the synchronization code for Memcached and detecting two potential correctness issues.

Distributed transactions on modern RDMA clusters promise high throughput and low latency. However, achieving good performance requires tuning the physical layout of the data store to the application and the characteristics of the underlying hardware. Manually tuning the physical design is error-prone, as well as time-consuming, and it needs to be repeated when the workload or the hardware change.

In Chapter 6 we quantify the effect of this tuning on the performance of OLTP workloads and present SPADE, a physical design tuner for OLTP workloads on modern RDMA clusters. SPADE automatically decides on the partitioning of data, tunes the index and storage parameters, and selects the right mix of direct remote data accesses and function shipping to maximize performance. To achieve this, SPADE combines information derived from the workload and the schema with low-level hardware and network performance characteristics gathered through micro-benchmarks. We implement and evaluate SPADE on top of FaRM [52, 53], and

show how tuning can achieve up to 2.2x higher performance over manual designs for two widely used OLTP benchmarks.

## 1.4 Thesis Statement and Contributions

This dissertation studies the effect of synchronization on applications that employ the shared-memory programming model. As we show, existing ways of understanding and extrapolating the scalability of applications developed for the shared-memory setting, as well as existing synchronization primitives, are workload agnostic and not tailored to the modern hardware landscape. In this dissertation, we propose using new sources of information to extrapolate the scalability of applications and show how synchronization primitives that consider workload and hardware characteristics can help shared-memory applications scale better.

In detail, this dissertation makes the following intellectual contributions:

- It introduces the use of stalled cycles in hardware and software for extrapolating the scalability of in-memory applications.
- It introduces a new locking paradigm, in which lock management and lock algorithm selection is abstracted from the programmer and handled by a locking runtime.
- It identifies and quantifies the effect of tuning On-Line Transaction Processing (OLTP) applications on modern RDMA clusters and shows how this tuning can be automated.

## 1.5 Roadmap

The rest of this dissertation is organized as follows:

- Part I provides some background on synchronization and modern hardware trends (Chapter 2) and discusses related work (Chapter 3).
- Part II presents a new technique for understanding and extrapolating the performance of in-memory applications (Chapter 4). It also shows how developers can identify the root cause of poor scalability in their applications.
- Part III shows how we can improve the scalability of applications that use two popular synchronization abstractions, locking and transactions. Chapter 5 proposes a locking runtime that improves the scalability of applications and minimizes the development effort for lock-based programming. Chapter 6 presents our analysis of the effect of tuning on OLTP applications running on modern RDMA-enabled clusters and introduces SPADE, a physical design tuner for OLTP workloads built in FaRM [52, 53].
- Part IV presents directions for future work and concludes this dissertation (Chapter 7).

# Part I

# Preliminaries

# 2 Background

In this chapter, we present some background related to this thesis. We begin by describing modern multi-cores and the trends that have affected them. We then continue by presenting some background on performance counters and stalled cycles, which form the basis of our work in Chapter 4. We continue by presenting lock-based programming, which we set out to improve in Chapter 5. Finally, we highlight recent hardware trends and how they affect distributed transactions, which is the topic of Chapter 6.

## 2.1 Modern Multi-Core Platforms

The shift of processor manufacturers from high-frequency, single-core processors in the early 2000s drastically changed the design of software. While before applications typically would benefit from the increase in the performance of processors, suddenly developers had to start developing multi-threaded applications, explicitly handling aspects of their applications such as data splitting and workload distribution. Even more, as more and more cores were added to processors, the single-thread performance of an application stopped being the most important performance metric, and *scalability*, or the ability of an application to use additional cores, became the most important performance characteristics of an application.

To better take advantage of the increasing transistor density on processors, manufacturers have added additional levels of cache memory, and additional processing units in each core. The additional levels of cache have been added to mitigate the growing gap between CPU and memory performance [73]. Nowadays, server processors typically have 3 levels of cache memory, with tens of MB of shared L3 cache memory per processor. However, multi-core processors need to solve a significant problem when caching is introduced: they need to ensure *cache coherence*, that is maintaining a consistent view of data, as it is modified by the private caches of different cores. To achieve that, they utilize coherence protocols, the most prominent of which is the MESI protocol [128].

Including more processing units in the same core is done to improve thread-level parallelism. While an executing thread stalls (e.g., because of data dependencies, or waiting on a memory transfer), more threads can run on the same core, using the additional processing units in it. In modern processors, such processing units are presented as *simultaneous multi-threading* (SMT) threads [162]. While SMT threads can execute in parallel, the fact that they share significant resources, such as caches and pipelines, means that they can also interfere with each other and degrade performance.

The last hardware trend in modern processors is regarding the interconnection of multiple processors in the same machine, to create *multi-processor* machines. Typically, each processor is housed in a different die, with a locally attached memory module. We refer to these configurations as *multi-socket* configurations. Manufacturers typically have specialized interconnects between sockets, such as Intel's QuickPath [80] and AMD's Hyper Transport [36]. The result of such configurations is that data accesses and shared data among cores residing on different sockets can incur higher latencies, thus limiting scalability. As such, taking NUMA effects into account becomes important, especially for machines with multiple sockets.

## 2.2 Stalled Cycles and Performance Counters

### 2.2.1 Stalled cycles

During the execution of an application, CPU cycles are spent to produce useful work, while part of the cycles is spent stalling (called *stalled cycles*), either at the hardware level (i.e., waiting on a cache line miss), or at the software level (i.e., spinning on a busy lock). In an ideal scenario, stalled cycles would not constitute a significant part of the execution time and cycles that produce useful work would be evenly distributed across processors, resulting in linear speedup for the application (assuming that the instructions executed do not change significantly when running on more cores).

However, this is rarely the case. Stalled cycles can comprise a significant part of execution time, increasing with the number of cores. Stalled cycles exist both in hardware and software. At the hardware level, stalls are the result of unavailability of processing units or data, which degrades the performance of an application as the number of cores increases. Parallel applications additionally need synchronization, which causes further increases in stalled cycles at the software level. These stalls minimize the benefits from scaling up an application and could be the reason behind even slowdown for higher core counts.

### 2.2.2 Hardware stalled cycles

Hardware stalls can be divided into two big categories, depending on the stalled execution stage. *Frontend stalls* are stalled cycles in the fetch and decoding phase of an instruction execution, while *backend stalls* are stalled cycles during the rest of the execution of an instruction.

Frontend stalls can typically be attributed to waiting on an instruction fetch that missed in the instruction cache, or a target fetch after a branch misprediction. Backend stalls are typically the result of resources or data not being available during execution. Both categories of stalled cycles have a negative effect on the performance of an application. However, frontend stalled cycles do not change significantly for increasing core counts. In contrast, backend stalled cycles have a direct impact on the scalability of an application, as they significantly increase when adding more cores.

### 2.2.3 Software stalled cycles

Previous research has shown that Instructions-Per-Cycle (IPC) related metrics are not always useful in predicting the performance of an application [11]. Stalled cycles are closely related to IPC calculations and can thus face the same problem. A processor can spend time executing instructions that do not contribute to useful work. If only hardware stalls are considered, this choice can be the source of prediction inaccuracies. Stalls at the software level can typically be found in synchronization of threads. This includes both lock-based synchronization (e.g., spinning on a busy lock), as well as optimistic concurrency control mechanisms, such as *software transactional memory (STM)* [74, 144]. In applications that use STM libraries, aborted transactions discard all work done inside the transaction.

Our work on ESTIMA ( Chapter 4) solves this problem by enabling the use of *software stalled cycles*. These represent cycles during which the application is executing instructions that do not produce useful work. Software stalls are optional: users can use a runtime that reports software-level stalled cycles, or modify their applications to provide such information, in order to improve the accuracy of predictions.

### 2.2.4 Extrapolating time

A straightforward approach for scalability predictions is to extrapolate the execution time of an application, measured for low core counts. Indeed, such approaches already exist [17] and provide high accuracy for the workloads they target. They typically function as follows: initially, they take measurements of the execution time of the application for different core counts. The next step is to use analytic functions to approximate the measurements, and extrapolate the measurements to higher core counts. An important drawback of this approach is that extrapolation requires scalability trends to be present in the existing measurements. When that is not the case, such as in the case of `kmeans`, shown in Figure 2.1, directly extrapolating time can lead to erroneous conclusions. In this case, the time extrapolation method predicts that the application will continue scaling for up to 48 cores, which is not the case. Similar cases can appear when small changes in the execution time steer the extrapolation towards wrong predictions. In Chapter 4) we show how ESTIMA compares against time extrapolation, as well as cases where ESTIMA is able to capture the scalability of applications for which trends are not visible in measurements.

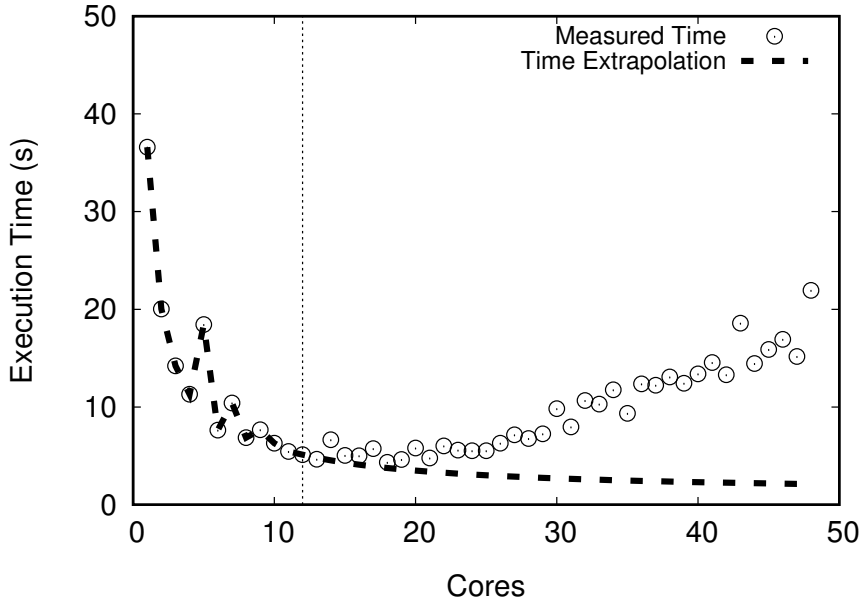



Figure 2.1 – Time extrapolation for `kmeans`.

## 2.3 Lock-based Programming

*Locks* are objects with two states: *busy* or *free*. A lock can be either free, or have a single *owner*, namely the thread that holds the lock. Locks ensure *mutual exclusion*: Only the owner can proceed with its execution, while any concurrent threads are waiting behind the lock. Locks offer two operations: *lock* and *unlock*. The former is used to *acquire* the lock (i.e., make the current thread the owner of that lock). Unlock *releases* the lock so that it can be subsequently acquired by another thread.

There are numerous lock algorithms. These algorithms mostly differ in the way they handle contention, namely the situation where threads are waiting for a busy lock to become free. Typical designs employ either *busy waiting*, or *blocking* for waiting. With busy waiting, waiting threads remain active, polling the lock until they manage to acquire it. With blocking, waiting threads release their hardware context to the OS. The OS is responsible for unblocking these "sleeping" threads when the owner releases that lock.

There exist various busy-waiting locking algorithms. Simple spinlocks, such as *test-and-set* (TAS), *test-and-test-and-set* (TTAS), and *ticket lock* [111] (TICKET), use a single memory location on which threads are busy waiting. Spinlocks are fast under low contention due to their simplicity. However, simple spinlocks might generate a lot of coherence traffic on the single memory location (i.e., cache line) of the lock [13]. Queue-based locks (e.g., MCS [111], CLH [37]) generate a queue of waiting nodes so that each thread is spinning on a unique location when busy waiting. Queue-based locks thus remove the single-memory-location bottleneck of simple spinlocks.

The most well-known blocking lock is the *mutex*-lock (MUTEX), part of the `pthread` library. Because the overheads of the OS for blocking and unblocking a thread are high, blocking locks typically employ a busy-waiting period before putting threads to sleep.

At a first glance, programming with locks looks simple. Locking is appealing precisely because of this simplicity. Nevertheless, using locks efficiently (i.e., achieving correct and fast designs) can become cumbersome, mainly because of various correctness and performance problems that are often associated with locks.

### 2.3.1 Programming with locks

In short, programming with locks involves the following steps:

1. Recognizing the various critical sections and the data they protect (i.e., the *granularity* of each lock).
2. Selecting and using concrete lock implementations.
3. Declaring the lock objects. Non-statically allocated locks must not only be declared, but allocated as well.
4. Initializing the locks.
5. Using the locks through their interface (i.e., *lock*, *trylock*, *unlock*) in order to protect the critical sections.
6. Destroying and possibly deallocating the locks.

If the developer implements any of these steps improperly, correctness and performance issues can emerge.

### 2.3.2 System correctness with locks

The most well-known bugs associated with locks are *deadlocks*. In a deadlock, a thread has acquired a lock $l_0$ and is waiting for an already acquired lock $l_1$, the owner of which is waiting for a different acquired lock $l_2$ and so forth. Finally, there is a thread that has acquired $l_n$ and is waiting for $l_0$, in which case none of the threads can make progress. Deadlocks are often a result of acquiring locks in the wrong order and are notoriously hard to debug.

Another hard-to-detect issue with locks is using uninitialized locks (i.e., trying to (un)lock a non-initialized lock object). This issue results in executions where the system may or may not work properly, depending on the initial value of the lock object's memory. Other common mistakes with locks are trying to lock the same object twice, unlocking a lock that is already free, or releasing a lock that has been acquired by another thread.[1] As with uninitialized locks, the latter issues can break the system.

[1] Releasing a lock owned by another thread, or acquiring a lock twice, can be also used as a feature.

These issues are common in practice [33]. In §5.3.2, we present an easy-to-use debugging extension of GLS for detecting all of these issues. In fact, we use GLS to detect and solve two of these issues in Memcached (§5.4.1).

### 2.3.3 System performance with locks

In addition to correctness problems, locks might become a performance bottleneck, mainly due to two different reasons.

First, highly-contended locks can easily become a bottleneck (e.g., the global lock in Memcached v1.4.13 [59] or in the Linux kernel [21, 104]). Removing these bottlenecks might require significant effort. The designer must redesign the critical sections and change the granularity of the locks. This process is of course prone to the correctness issues discussed earlier.

The second and more important reason is that different lock algorithms are suitable for different workloads [40, 66]. As we show in Figure 5.1, simple spinlocks shine under very-low contention, queue-based locks are by far the best under medium to high contention, and blocking locks are necessary under multiprogrammed workloads. Choosing the "wrong" algorithm under multiprogramming can lead to *livelocks*, situations where although the threads execute, the system throughput is close to zero (see MySQL in §5.4.2).

Naturally, while designing and implementing a general-purpose system, the designer cannot predict every single deployment or potential runtime fluctuation of the behavior of the workload. Therefore, she must choose the common-denominator lock algorithm (i.e., the one that works even on multiprogramming), namely mutex. Unfortunately, studies have shown that mutex is slow compared to other algorithms in the absence of multiprogramming [40].

## 2.4 Distributed Transactions and Modern Hardware

### 2.4.1 Modern hardware trends

Modern hardware clusters follow two main trends: large main memories and fast network interconnects. Indeed, commodity machines nowadays typically contain a few hundreds of gigabytes of main memory at very low prices. This enables handling tens of terabytes of data with only a small cluster of machines, benefiting from the low latency and high throughput of main memory and avoiding complex buffering mechanisms [149].

Ensuring data durability in main memory requires persistent logging. In order to avoid the overheads of logging to disks or SSDs, machines of a cluster can be equipped with a "distributed uninterruptible power supply" (distributed UPS) [53, 114] that writes the contents of the memory to an SSD when a power failure occurs, effectively making all DRAM non-volatile. New Non-Volatile RAM (NVRAM) technologies that have been proposed [99, 113, 137, 150] provide an alternative, but have not been widely deployed yet.

At the same time, datacenter networking has also evolved. Commodity Network Interface Controllers (NICs) nowadays support Remote Direct Memory Access (RDMA). Current implementations include InfiniBand, RoCE (RDMA over Converged Ethernet) and iWARP (internet Wide Area RDMA Protocol). Connection-based implementations support one-sided RDMA requests. Applications register memory regions with the NIC, after which the NIC can serve one-sided RDMA requests (e.g., remote read or write requests) without involving the CPU. Remote reads and writes are currently only supported in reliable connection-based implementations. RDMA has been exploited to increase performance in a number of different systems and scenarios [18, 79, 89, 100, 116, 146, 151, 161].

### 2.4.2 Distributed transactions

Distributed database transactions are old, but still an active area of research [90, 155, 170]. Recently, various systems have utilized fast networks and modern hardware to achieve millions of OLTP transactions at the sub-millisecond scale [8, 32, 52, 88, 106, 174]. Most of these systems either build (or rely on) message-passing abstractions (typically Remote Procedure Calls – RPCs), utilize one-sided RDMA operations to directly access the memory of remote machines, or use a combination of the two (e.g., one-sided RDMA reads and writes with RPCs).

There are two main execution models for distributed transactions. In the first [32, 52, 88], often referred to as *symmetric*, all the machines participate in storing data and executing transactions. Each server is responsible for storing parts of the data (and possibly replicas of the data stored by other machines of the cluster), as well as running distributed transactions. The second model [106, 174] distinguishes between computation and storage nodes. The former run transactions, utilizing RDMA to access the data stored in the memory of the latter.

We use FaRM[52, 53] as our target platform. In FaRM, objects are stored in the memory of the cluster with a global address. FaRM uses one-sided RDMA accesses for lock-free reads, as well as message-passing for read-write transactions. It offers an API that allows applications to create and execute transactions. It also includes implementations of a distributed hashtable and a distributed B-Tree.

### 2.4.3 The cost of performance on modern RDMA networks

Distributed transaction systems typically offer APIs for low-level operations within transactions. Some of them also include implementations of data structures that can be used for OLTP workloads. More specifically, [8, 32, 52, 88, 106, 174] have a hashtable implementation, while [32, 52, 106, 174] have a B-Tree [35] implementation. Even with the available APIs and data structures, implementing OLTP workloads using such an API is not straightforward. For example, given only two of the queries of TATP benchmark [120], *GET_SUBSCRIBER_DATA* and *UPDATE_LOCATION* (shown in Listing 2.1 and Listing 2.2), there are a few crucial choices to get performance out of the underlying hardware. For the *Subscriber* table, these are:

1. Table storage. This includes data partitioning and deciding on how the data is stored, for example inlining data in an index, or flat storage of data.
2. Indexes. By examining the queries, in addition to a primary hash index on *s_id*, a secondary hash index on *sub_nbr* is necessary. Based on the choices in the previous step, each index needs additional tuning to get the best performance of the underlying network, based on characteristics such as the record size, the operation mix and the neighbourhood size of the hash index.
3. Remote data access. Possible options here include: a) remotely accessing the data using RDMA, b) doing an RPC to the server holding the data, or c) a combination of the two. Choosing when to remotely access data and when to send a message can be fine-tuned based on the length of the operations and the size of transfers necessary.

```
SELECT s_id, sub_nbr,
       bit_1, bit_2, bit_3, bit_4, bit_5,
       bit_6, bit_7, bit_8, bit_9, bit_10,
       hex_1, hex_2, hex_3, hex_4, hex_5,
       hex_6, hex_7, hex_8, hex_9, hex_10,
       byte2_1, byte2_2, byte2_3, byte2_4,
       byte2_5, byte2_6, byte2_7, byte2_8,
       byte2_9, byte2_10,
       msc_location, vlr_location
FROM Subscriber
WHERE s_id = <rnd>;
```

Listing 2.1 – TATP's *GET_SUBSCRIBER_DATA* query.

```
UPDATE Subscriber
SET vlr_location = <rnd>
WHERE sub_nbr = <rnd>;
```

Listing 2.2 – TATP's *UPDATE_LOCATION* query.

Making the correct choices is not only cumbersome, but also error prone. In fact, workloads typically contain a mix of operations for each table and queries can touch data on more than one tables (and possibly machines). Deciding on a set of parameters for each table, based on the workload and the hardware it will execute on quickly becomes overwhelming, and making the wrong choices can have a significant impact on performance, as we showed in Figure 6.1. Additionally, changing the workload requires repeating the entire process, which is time-consuming if done manually. Previous work has implemented and studied the performance of OLTP transactions on modern RDMA clusters, but has not looked at the trade-offs of tuning, or quantified the effect that these choices have on the performance of an OLTP workload on modern RDMA networks.

# 3 Related Work

In this chapter we present related work to the topics addressed by this dissertation. We begin with work related to our scalability prediction tool (presented in Chapter 4) in Section 3.1. We then present work on locking for multi-cores and adaptive locking in Section 3.2, which relates to our work in Chapter 5. Finally, in Section 3.3, we describe work in the area of distributed transactions on modern hardware, which relates to our work presented in Chapter 6.

## 3.1 Scalability Prediction and Performance Analysis

The work that is most closely related to ESTIMA (Chapter 4) is that of Crovella et al. [38], in which the authors identify two categories of stalls at the software level: productive cycles and stalled cycles. They collect measurements of the two categories of cycles and use them to predict the performance of an application. As we show in Chapter 4 using low-overhead performance counters offered by modern hardware can reduce overheads. At the same time, adding software-level stalls to measure synchronization overheads can further improve accuracy, making a solution applicable to a wider variety of workloads.

Various research projects have studied the scalability of parallel and distributed applications. Barnes et al. [17] use linear logarithmic functions to predict the scalability of message-passing scientific applications on large-scale parallel systems. The number and the configuration of CPUs are the inputs and the process uses linear logarithmic functions. In [98], statistical techniques and regression analysis are used to build piecewise black-box polynomial and neural network models of scientific programs. Neural networks are also used in [83] to build models of SMG2000 applications executing on two different large-scale machines. In [124], the author extrapolates address stream profiles, using low-level metrics collected on memory access patterns, in order to study the memory performance of an application under strong scaling. Finally, in [34], the authors use call path profiles and expectations on the cost differences between executions to estimate the scalability costs incurred by different parts of the program. Our work in Chapter 4 focuses on the scalability of in-memory applications, on

machines with significantly higher number of CPUs available, and uses both hardware and software stalls to extrapolate the scalability of the application as a whole.

Several systems combine predictions of the sequential performance of single-node tasks performed by distributed cores with models of communication between them [27, 109, 175]. Similar cross-platform performance predictions for large-scale machines using a combination of known relative performance of the two systems and partial execution of the workload are described in [172]. Various formal modeling techniques for distributed and concurrent systems have also been proposed [84], including petri nets and queuing theory [131]. They were used to develop detailed analytic models for several applications [78, 93]. These models are very accurate. They require however in-depth understanding of the applications and the system. In contrast, ESTIMA (Chapter 4) can be used with little effort on *any* parallel in-memory application.

Models based on discrete-time Markov chains were developed for several STM algorithms [69, 70, 71] to compare different STM designs. Usui et al. [163] use a simple cost-benefit analysis to choose between locking and transactions. The performance model from [134] focuses on modeling transactional conflict behavior. Unlike ESTIMA, this approach requires heavy instrumentation of the application memory accesses.

Performance counter research has mainly focused on profiling of applications. In [147] and [164], the authors use performance counters to increase power efficiency, through thread scheduling and placement. In [166], the authors use performance counters to capture performance impacts as a function of resource usage. Srikanthan et al. [148] use performance counters for online workload scheduling in multiprogrammed environments. Jiménez et al. [85] devise a model based on performance counters to predict total power consumption. While performance counters have been used for many different goals, the low-level information they provide has not yet been exploited for scalability predictions, as in ESTIMA.

Finally, numerous projects have focused on identifying bottlenecks in parallel applications. Liu et al. [105] develop ScaAnalyzer, a tool that uses event-based sampling to identify memory-related bottlenecks in parallel applications and suggest optimizations that improve scaling. In [173], the author proposes a set of new performance counters, which, together with a new analysis method can identify performance bottlenecks in out-of-order processors. Torrellas et al. [156] use hardware performance counters to identify scalability bottlenecks in parallel applications running on distributed shared-memory multiprocessors. The main goal of our work in Chapter 4 is to predict the scalability of an application for larger machines. By pinpointing the sources of stalled cycles predicted by ESTIMA, developers can also identify bottlenecks that will appear in their applications. This, however, does not replace tools specifically built for performance tuning and bottleneck identification.

## 3.2 Locking for Multi-Core Processors

**Lock algorithms.** Apart from the traditional spinlock algorithms (e.g., test-and-set, ticket locks [111]), there are several efforts towards designing more scalable locks. Mellor-Crummey et al. [111] and Anderson [13] introduce several alternatives, such as queue-based locks. Luchangco et al. [108] design a hierarchical CLH lock [37] for NUMA architectures. Dice et al. [46] generalize the design of NUMA-aware hierarchical locks by introducing a technique for converting any lock algorithm to be hierarchical. Chabbi et al. [28] and Zhang et al. [176] design locks for NUMA systems, delivering performance in various contention scenarios. David et al. [40] analyze various lock algorithms on different platforms and point out that "every lock has its fifteen minutes of fame." Our GLS middleware and GLK algorithm ( Chapter 5) directly build on top of the observations and the results of such prior work. Concurrently with our work, Guiroux et al. [66] study the performance of lock algorithms in applications and confirm the previous findings. Their LiTL library uses a similar approach to GLS, but with the aim of easily switching between lock algorithms.

**Adaptiveness in locks.** Karlin et al. [92] analyze the cost of busy waiting and blocking and show that adaptive techniques (i.e., adapting the amount of spinning before threads block) deliver the best performance. Similarly, Falsafi et al. [57] design a mutex algorithm that dynamically adapts the amount of spinning when locking/unlocking, to achieve higher energy efficiency. Many modern lock designs, such as the classic mutex lock, both on Linux [1] and on Solaris [153], include this type of adaptiveness. Similarly, the Hotspot 7 JVM modifies its semaphore algorithm at runtime to save memory, based on the idea of thin locks [16]. If there are no threads waiting behind the lock, the JVM represents the lock with just a few bits, otherwise it converts the lock to keep track of the queue of waiting threads. Lock elision [60, 138, 139] aims at reducing the overhead of locking when critical sections do not actually conflict. A thread can optimistically execute its critical section without acquiring a lock. If a data conflict appears, then the thread rolls back and executes the critical section normally. Diniz and Rinard [47] introduce dynamic feedback, where the compiler generates various synchronization policies that can then be selected at runtime based on sampling.

Lim and Agarwal [101] propose reactive locks, an adaptive synchronization scheme that switches between different protocols and waiting strategies. In this sense, GLK can be viewed as an instance of reactive locks. However, GLK is a fully practical adaptive lock algorithm, tailored to modern multi-cores. GLK includes three concrete lock algorithms which cover the needs of modern software systems, tuned approaches for collecting contention statistics for the locks, as well as a low-overhead method for detecting multiprogramming and switching to a blocking lock when necessary.

Johnson et al. [87] present a locking technique (LC) that decouples waiting from thread scheduling. A global monitor controls how many threads must block, while the remaining threads use time-published MCS locks [68]. The background thread we use in GLS is similar to

this global monitor. However, in our work, every lock object is a "normal" lock with just a hint on whether it should employ the *mutex* mode.

**Debugging locks.** Debugging locks is notoriously hard. The main techniques available today can be categorized into *static* and *dynamic*. Static techniques identify deadlocks by analyzing the lock and flow graphs of an application and by employing heuristics for identifying the prerequisites for a deadlock [55, 119, 167]. Dynamic deadlock-detection techniques employ ways of monitoring the acquired locks for each thread and discovering cycles in the locks that are being locked [94, 135, 142, 160]. More specifically, Koskinen et al. [94] implement a deadlock detection algorithm that allows lock operations to expire. Pyla et al. [135] implement a runtime for deadlock detection for applications that use `pthreads`. Samak et al. [142] use dynamic analysis and execution traces to identify possible deadlocks. GLS includes a low-overhead, dynamic debugging mode, which can identify issues in lock-based applications.

**Alternatives to traditional locks.** Transactional memory, in software (STM) [51, 117, 145] or in hardware (HTM) [45, 75], replaces locks with transactions as a concurrency-control mechanism. On the one hand, STMs are typically slower than locks, due to their instrumentation overhead. On the other hand, HTMs are not mature enough and cannot yet fully replace locking [169]. Flat combining [72] is a technique for optimizing coarse-grained locks (e.g., the global lock of a queue). With flat combining, a critical section translates to a message to a dedicated server thread that executes the request on behalf of the invoking thread. Similarly to flat combining, RCL [107] overloads the lock function for highly-contended locks with remote procedure calls on a dedicated server thread. Unlike flat combining and RCL, GLK optimizes both lightly- and highly-contended locks.

**Locks in systems.** There are various efforts in operating systems to minimize sharing [24], to remove lock-related bottlenecks [25, 26], or to completely avoid locks [15, 19] in order to resolve scalability bottlenecks. Similarly, in data stores and DBMSs, recent projects [48, 59, 65, 86, 107, 110] show significant performance improvements in systems such as Memcached and RocksDB, mainly by removing contended locks. GLS is designed to make development of such systems easier, while achieving high performance.

## 3.3 Distributed Transactions in the Datacenter

### 3.3.1 Distributed transactions and modern hardware

Distributed transactions on traditional hardware and networks has a long history of research. However, in the past, networks imposed a significant bottleneck that dominated performance. As such, a large body of research has focused on partitioning schemes [39, 126, 129, 136, 143] to avoid crossing partitions, or relaxation of the consistency of transactions [95].

Recently, a number of research projects and commercial systems have taken advantage of modern network characteristics, such as high throughput and low latencies, as well as RDMA capabilities, to implement fast distributed transactions. FaRM [52, 53] uses one-sided RDMA verbs and optimistic concurrency control to implement serializable distributed transactions. FaRM uses primary-backup replication to ensure fault-tolerance, a fast failure detection mechanism, and a recovery protocol that ensures that transactions are not blocked when failures occur. FaSST [88] implements serializable distributed transactions on top of unreliable, unordered, non-congestion-friendly RPC implementation. FaSST also uses primary-backup replication and a similar optimistic commit protocol. Tell [106] decouples processing nodes from storage nodes, using one-sided RDMA accesses to ship data to processing nodes. Tell implements a distributed snapshot isolation protocol that uses a centralized commit manager. NAM-DB [174] also separates computation from storage, however it uses a distributed snapshot isolation algorithm that relies on a scalable timestamp oracle. DrTM+R [32] provides serializable transactions by using both RDMA accesses and Hardware Transactional Memory (HTM), relying on primary-backup and an optimistic replication scheme.

All the above systems [32, 52, 53, 88, 106, 174] take advantage of modern hardware to implement distributed transactions. Moreover, they all implement some type of distributed index structures. In Chapter 6 we present SPADE, which optimizes for the physical design of data, such as tuning indexes and remote accesses, which is applicable to distributed systems that leverage modern RDMA networks in general.

### 3.3.2 Index tuning

Databases require a significant amount of tuning to achieve performance. Research on automatically tuning indexes has been meticulous [7, 64, 91, 130, 165]. Additionally, DBMS vendors provide tools that aid administrators in tuning a database. Microsoft's SQL Server has the Database Tuning Advisor [5], which can both tune existing indexes for performance, as well as recommend new ones. Oracle Server has both self-tuning [20, 44, 177] and performance analysis capabilities [171]. DB2 has offered the Performance Wizard tool [96]. Recent research projects have also looked into machine learning [10], adaptive sampling [54] and regression models [157] to choose the best parameters for DBMSs.

In Chapter 6 we focus on a distributed setting. We argue that, in addition to previous efforts for tuning of indexes, modern transactional systems require an additional level of tuning. SPADE takes into account the network and hardware characteristics to tune index parameters. This way, applications take full advantage of the performance of modern networks, which, as we show, can have a significant impact on performance.

### 3.3.3 Partitioning and remote accesses

Partitioning has played a role of paramount importance in the past for distributed transactions. Schism [39] is an off-line approach to data partitioning for distributed shared-nothing databases, which minimizes the number of distributed transactions. SWORD [136] employs an incremental data repartitioning technique for OLTP workloads in database-as-a-service cloud settings. Pavlo et al. [129] automatically partition workloads using large neighborhood search and analytical models. Clay [143] is an on-line partitioning approach that uses dynamic blocks to incrementally partition data minimizing the number of distributed transactions. JECB [158] uses a divide-and-conquer strategy to partition workloads for large clusters. Sun et al. [152] use a bottom-up approximate approach to partition data.

With the advent of fast networks and one-sided remote operations, partitioning is no longer the determining factor, but rather an optimization. With new ways to access remote data, partitioning now imposes a new problem, that of determining how to do so in a way that takes full advantage of the network characteristics. As such, SPADE's optimization of remote accesses plays an important role, as we show in our evaluation.

# Part II

# Understanding the Scalability of Shared-Memory Applications

# 4 Extrapolating the Scalability of In-Memory Applications

## 4.1 Introduction

Commodity machines nowadays have hundreds of gigabytes of memory. This enables building performance-critical parallel applications, such as databases and key-value stores, that keep their datasets in main memory. This way, applications avoid slow secondary storage and networks, leaving the CPU as the main performance bottleneck [43, 58, 102, 121]. Understanding the performance of these applications proves to be hard, since the number of CPU cores available during the deployment of a parallel application can be significantly higher than that during its development and testing. Applications developed today can be tested on machines with 16 or 24 cores, but in a few years the same applications are likely to be run on machines with 64 or even more cores.

Consequently, the crucial question about performance of in-memory applications is that of their scalability on an increasing number of cores. Answering this question is very hard. Typical approaches include performing extensive performance evaluation or developing detailed models of applications [70, 109], which are time-consuming, error-prone, and require detailed knowledge of each application and the machine it executes on.

In this chapter we present ESTIMA, a practical tool that enables developers and users to predict the scalability of parallel in-memory applications in a simple way, without having to understand in detail the internals of the application or the machine it will run on. ESTIMA enables developers to visualize the scalability of their applications, as well as to discover bottlenecks that might not be evident during initial performance benchmarking. ESTIMA can be applied with little effort to *any* parallel in-memory application, in contrast to other approaches that heavily rely on application-specific information [17, 27, 84, 93, 98, 122, 175].

Instead, ESTIMA leverages *stalled cycles* to extrapolate the scalability of an application. These are cycles the application spends on non-useful work, such as waiting for a cache line to be fetched from memory or waiting on a busy lock. Contention for shared resources typically increases with the number of cores used by an application, resulting in an increase in stalled

cycles that directly impact the application's scalability. The application's performance keeps improving as long as adding cores mainly increases the number of useful cycles. As soon as adding more cores mostly results in stalls, performance stops improving, or even degrades: the application stops scaling.

ESTIMA measures stalled cycles in both hardware and software and extrapolates them (using analytic functions) to higher core counts to predict the overheads of using more cores. Then, ESTIMA correlates stalls to execution time in order to produce predictions of the execution time of the application at higher core counts. In addition to predicting scalability, analyzing the dominating stalled cycle categories reported by ESTIMA can reveal bottlenecks that will appear for higher core counts and guide developers' optimization efforts. To the best of our knowledge, ESTIMA is the first system to use stalled cycles for scalability extrapolations.

By default, ESTIMA uses hardware performance counters to measure hardware stalls. These are counters offered by modern hardware that can collect the values of events that stall the execution of an application with low overhead. Measuring software stalls requires configuring or instrumenting runtime libraries, such as `pthreads` or a transactional memory library. In our experience, software stalls can be exposed with minimal changes to the runtime libraries, but because they are not always available, ESTIMA does not require software stalled cycles to function.

Our evaluation shows that ESTIMA's simple approach yields accurate predictions. We illustrate the use of ESTIMA to successfully predict the performance of a `memcached` and a `SQLite` workload on a server machine based on measurements on a desktop. We then extensively evaluate ESTIMA using 21 workloads that span a wide range of application characteristics and synchronization techniques on two different platforms: a 4-socket, 48-core AMD *Opteron* machine and a 2-socket, 20-core Intel *Xeon* machine. We conduct both strong scaling and weak scaling experiments. Finally, we pick two applications that exhibit poor scalability (`streamcluster` and `intruder` from our benchmark workloads) and show how stalled cycles can help identify bottlenecks. More specifically:

- ESTIMA accurately extrapolates scalability between different machines with similar architectures on real-world workloads. For our `memcached` and `SQLite` workloads measured on a desktop machine, ESTIMA predicts their scalability on a server machine with errors lower than 30% and 26% respectively.
- ESTIMA successfully captures the scalability of all applications we consider for its evaluation, correctly identifying the number of cores for which the applications stop scaling, for both strong and weak scaling predictions. The predictions are fairly accurate in absolute terms too. ESTIMA can predict performance when doubling the number of cores with errors lower than 15% on more than half of the workloads.
- Prediction errors do not result in prediction of a different behavior. In other words, there are no cases where ESTIMA predicts that an application will scale, when in fact it does not.

- ESTIMA can help identify bottlenecks as we illustrate through two parallel applications that exhibit poor scalability, `intruder` and `streamcluster` from the STAMP and PARSEC benchmark suites respectively.

In summary, the main contribution of this work is ESTIMA, a practical tool that uses stalled cycles from both hardware and software sources to extrapolate the scalability of in-memory applications. ESTIMA can help users make better decisions regarding the deployment of their application. To the best of our knowledge, ESTIMA is the first system to use stalled cycles for scalability extrapolations. It showcases the power of stalled cycles in extrapolating the scalability of an application, while being applicable to a wider range of application than existing tools and methodologies.

The rest of this chapter is organized as follows. We present the insights behind ESTIMA in Section 4.2. We explain how ESTIMA works in Section 4.3. We present the implementation, as well as our extensive evaluation of ESTIMA in Section 4.4. We discuss some interesting findings of our work in Section 4.5 and conclude the chapter in Section 4.6.

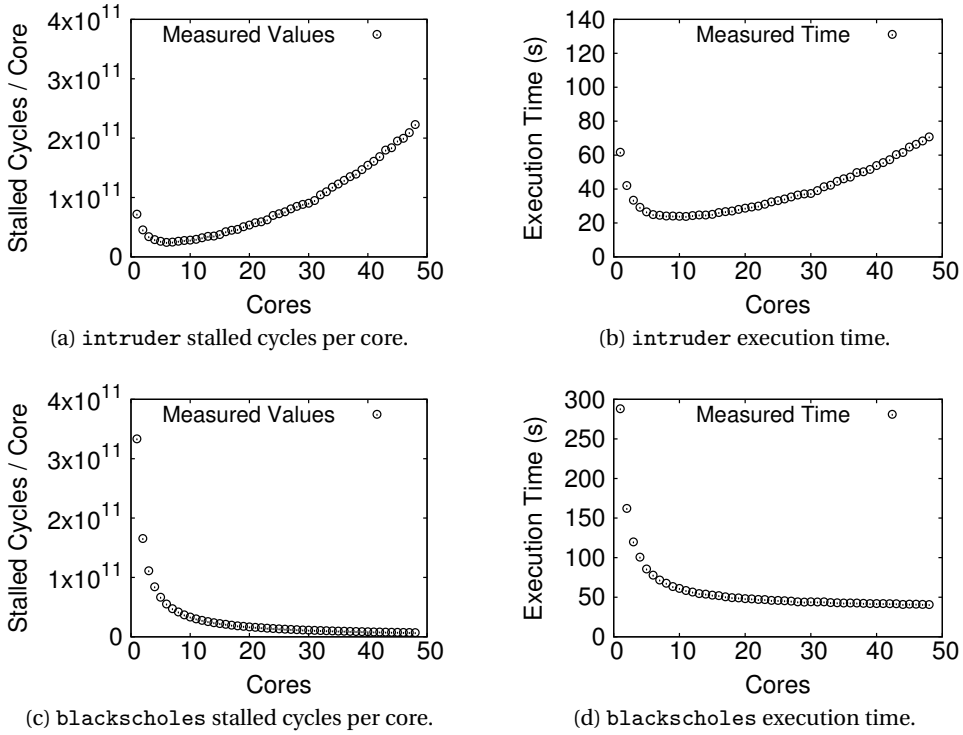
## 4.2 ESTIMA: Insights

In this section, we present the insights behind ESTIMA. We first recall what stalled cycles are, their main sources, and how they affect scalability. We discuss why we use stalled cycles in ESTIMA, in contrast to the straightforward approach of extrapolating time. Finally, we present some of the decisions we have made when designing ESTIMA and how they affect its capabilities.

### 4.2.1 Stalled cycles for scalability predictions

ESTIMA uses the number of stalled cycles per core to predict the scalability of an application as the number of cores increases. We show two examples of applications, presenting the stalled cycles per core and the execution time in Figure 4.1. They are the `intruder` and `blackscholes` benchmarks from the STAMP [115] and PARSEC [22] suites respectively. For both applications, the correlation of the number of stalled cycles per core to execution time is 1.00. We evaluate the correlation of stalled cycles to execution time more extensively in Section 4.5.1.

ESTIMA leverages stalled cycles for its predictions. By default, ESTIMA uses *hardware performance counters*, dedicated CPU registers, used to monitor performance events, such as the number of instructions executed, cache misses per cache level, stalled cycles, as well as I/O requests and memory accesses. Different architectures offer various performance counter events that measure a wide range of backend stalled cycles. There usually exist aggregate events that measure the total backend hardware stalled cycles, as well as more detailed events that measure different types of backend stalled cycles individually [12, 82]. ESTIMA does not




(a) `intruder` stalled cycles per core.

(b) `intruder` execution time.

(c) `blackscholes` stalled cycles per core.

(d) `blackscholes` execution time.

Figure 4.1 – Stalled cycles and execution time correlation.

use the aggregate events. Instead, it uses the performance counters that measure fine-grain backend stalled cycles on each architecture. These counters are low-level enough to provide insights into the behavior of the application for higher core counts. In addition, when combined, they give the same high-level image of the scalability of the application as aggregate events.

The insight behind using fine-grain stall events is the following: using an aggregate event would be similar to extrapolating the execution time itself, since the two follow the same trends. For example, from the aggregate backend stalls shown in Figure 4.1, with measurements up to 12 cores, we do not see trends that show the poor scalability of the applications for higher core counts. Using aggregate events would not capture significant changes in the scalability of applications, which is the main goal of ESTIMA. By using fine-grain stalls, trends appear in their values for lower core counts, before the effect on the scalability of the application is significant. These trends are helpful in predicting scalability changes that are otherwise difficult to capture (we give more details in the prediction example in Section 4.3.2). A second reason for using the individual stalled cycles is that aggregate events do not provide any information on the scalability bottlenecks. By using the detailed events and pinpointing the parts of the code they come from, ESTIMA can help identify the area that inhibits scalability, as we show in Section 4.4.6.

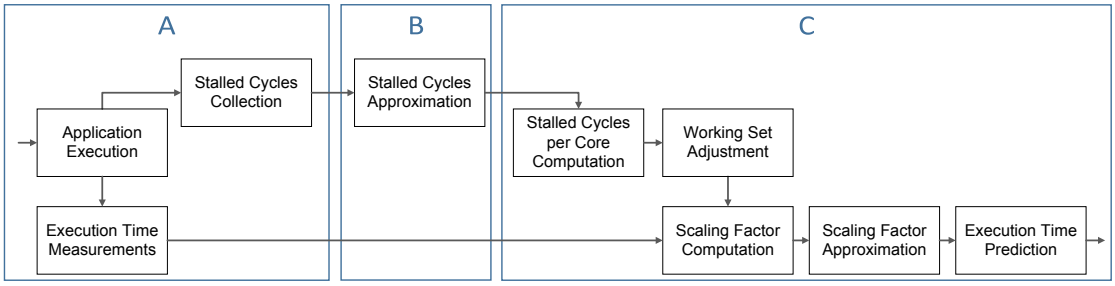



Figure 4.2 – Constructing extrapolations with ESTIMA.

### 4.2.2 Other performance counters

Prior work [166] has used performance counters to measure events such as cache misses and branch mispredictions to identify bottlenecks in applications. A similar approach in ESTIMA would involve extrapolating counters such as cache misses for the different levels of cache and incorporating them in the prediction process. The problem with this approach is that cache misses require a very detailed model that takes into account the memory access patterns of the application. This is necessary in order to translate the misses captured to time and quantify their effect to the scalability of the application. Quantifying the interactions between misses and scalability requires detailed knowledge of the application, which is against the generic purpose of ESTIMA. For this reason, we chose not to use cache misses in our predictions. However, their effect is captured by the stalled cycles that ESTIMA uses. In this case, their actual effect on scalability is captured through its manifestation in stalls in the pipeline.

## 4.3 ESTIMA

The prediction scheme of ESTIMA is depicted in Figure 4.2. It involves three main steps: (A) first, ESTIMA executes the application on the *measurements machine*, collecting different types of stalled cycles from hardware and optionally from software. (B) Then, ESTIMA extrapolates the values of these stalls to higher core counts, using regression analysis and a set of pre-defined function kernels. (C) Finally, ESTIMA combines the extrapolated values and calculates the stalled cycles per core. By correlating stalled cycles to execution time, ESTIMA predicts the execution time of the application for higher core counts. In the next sections, we provide a detailed description of the internals of this process and present a step-by-step execution example of ESTIMA.

### 4.3.1 Prediction process

**Stalled cycles collection.** The first step of the prediction process of ESTIMA is to execute the application for different core counts, up to the number of cores available on the measurements machine, collecting hardware performance counters and software-reported stalls. During the

execution of the application, ESTIMA also measures the application's execution time. ESTIMA uses these measurements for the execution time predictions in the last step of the process.

For hardware stalls, ESTIMA collects the backend stalled cycles (as available by the architecture). Choosing the backend stalls for an architecture involves identifying the counters that measure stalled cycles in the pipeline. From this set, we discard the events that refer to instruction fetching, keeping only the stalls in the execution phase of an instruction. We also discard events that significantly overlap, such as aggregate events for backend stalls. Intuitively, stalls that overlap can make predictions pessimistic, depending on the extent to which they overlap.

Processor families typically share the same set of counters, and using ESTIMA with different processors of the same family requires no changes in its configuration. Adding support for a new processor family requires consulting the developer's manual of the processor and identifying these backend stalls. For the machines that we had available, identifying the stalls to be used was a simple task that was necessary only once for each manufacturer. The same counters were then used for both desktop and server machines, without the need to choose different counters.

When choosing the software stalls to be (optionally) collected, the developer needs to consider the parts of (mainly synchronization) code that, when executed, produce no useful work for the workload. Such cases include (but are not limited to) spinning on locks and looping on

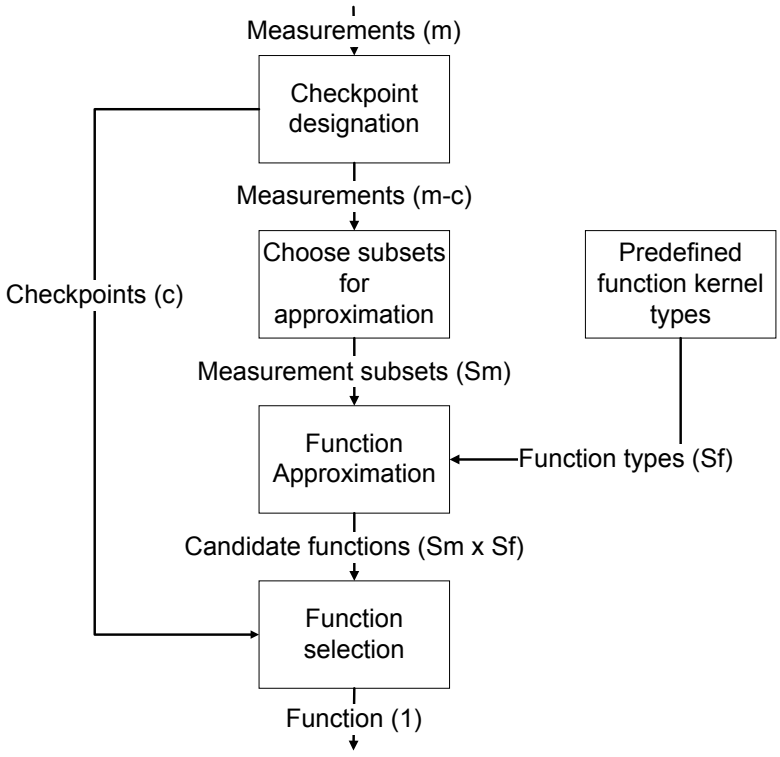



Figure 4.3 – The regression analysis of ESTIMA.

*trylock* operations. An interesting case is *Software Transactional Memory*, where deciding on the cycles that are not producing useful work is straightforward (aborted transactions), and an STM runtime can report these measurements directly.

**Stalled cycles regression analysis.** This step consists of regressing the stalled cycles measurements. ESTIMA uses function approximation [4] to construct a set of functions for each stall category. Function approximation takes the measurements, a function type (e.g., a polynomial function of degree $d$) and constructs a function that closely fits the measurements (e.g., calculates the coefficients of the polynomial). ESTIMA chooses one function for each stall category and uses it to extrapolate the measured values. The approximation process, shown in Figure 4.3, consists of the following steps, assuming $m$ measured values for a specific category of stalled cycles:

1. From the $m$ available measurements, ESTIMA designates the $c$ measurements with the highest core counts as checkpoints. In our experiments, we set $c$ to 2 and 4.
2. Using the first $n$ measurements ($n = m - c$), ESTIMA creates functions from the predefined kernels in Table 4.1. These functions are used based on the approximation library used (see Section 4.4.1), discarding the function types that produce functions that are not realistic for this approximation. The process is repeated for $i$ in $3..n$, to avoid over-fitting the function to the available measurements. Intuitively, small deviations in the measurements sometimes steer the function in the wrong direction, resulting in less accurate predictions.
3. For each of the constructed functions, ESTIMA calculates the root mean square error (RMSE) at the checkpoints. By using only the checkpoints, functions that have deviations for low core counts but approximate performance counter values accurately for higher core counts are considered as possible choices.
4. ESTIMA chooses the function that minimizes the error and subsequently uses it to approximate the stalled cycle values for higher core counts.

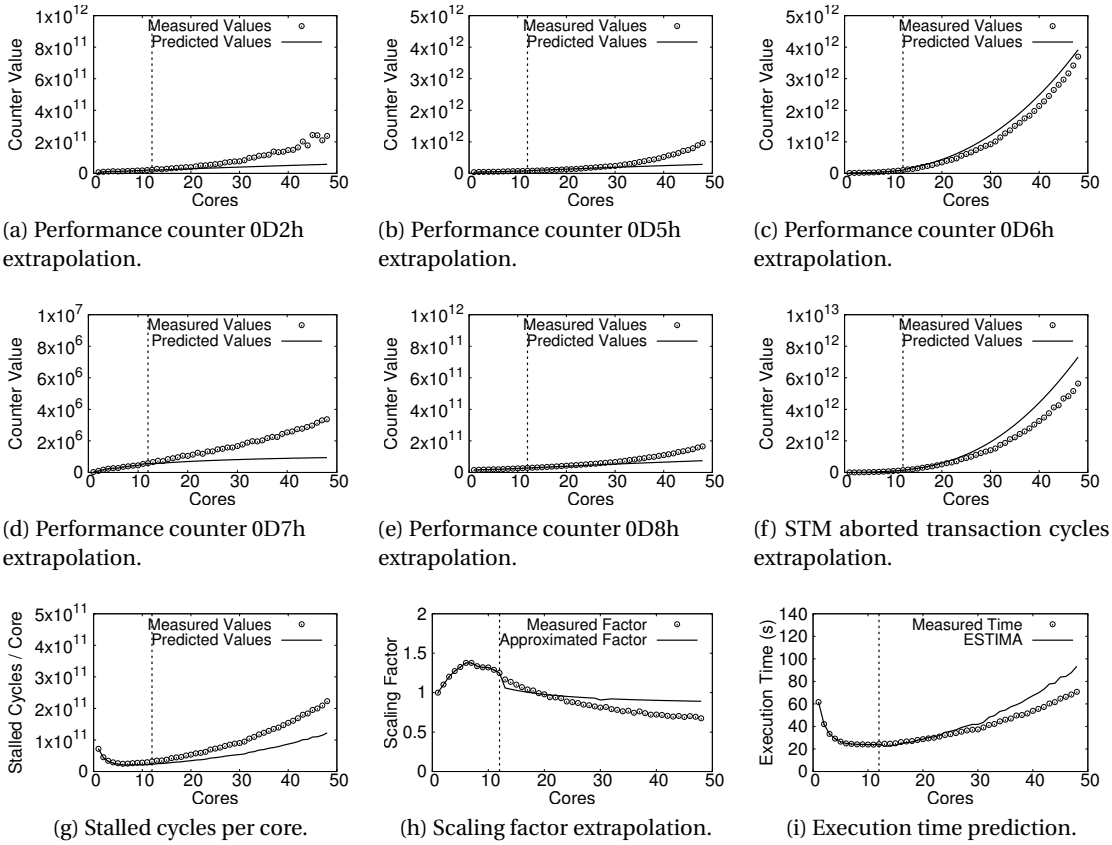
**Translating stalled cycles to execution time.** After all the stalled cycle events have been approximated, ESTIMA calculates the total stalled cycles per core, using the approximated values. The total stalled cycles per core and the execution time have similar curves, including

Table 4.1 – Extrapolation function types.

| Name | Function |
|---|---|
| *Rat22* | $\frac{a_0+a_1 n+a_2 n^2}{1+b_1 n+b_2 n^2}$ |
| *Rat23* | $\frac{a_0+a_1 n+a_2 n^2}{1+b_1 n+b_2 n^2+b_3 n^3}$ |
| *Rat33* | $\frac{a_0+a_1 n+a_2 n^2+a_3 n^3}{1+b_1 n+b_2 n^2+b_3 n^3}$ |
| *CubicLn* | $a+b\ln(n)+c\ln(n)^2+d\ln(n)^3$ |
| *ExpRat* | $e^{\frac{a+bn}{c+dn}}$ |
| *Poly25* | $y=a+bx+cx^2+dx^{2.5}$ |

minima and maxima points. However, they represent different quantities. An example has already been introduced in Figure 4.1, where execution time and stalled cycles are shown for the `intruder` and `blackscholes` applications. The two quantities are not *similar*, in the sense that there is no constant number that connects them. Their similarity factor is rather a function of the number of cores.

ESTIMA uses the stalled cycles and the execution time measurements, collected during the execution of the application, to calculate the values of the scaling factor function for the available core counts. It then extrapolates this function using the same kernels as before (Table 4.1). In this case, ESTIMA no longer chooses the function that best fits the points. In contrast, it chooses the function that produces execution time predictions that have the highest correlation to the total stalled cycles per core. The reason is that we have already argued that execution time and stalled cycles have a very high correlation. As such, the produced execution time values should retain high correlation to the total stalled cycles per core that were calculated in the previous step. After the factor function has been created, ESTIMA uses the stalled cycles per core (from step B) and the factor function to calculate the execution time of the application for higher core counts.




(a) Performance counter 0D2h extrapolation.

(b) Performance counter 0D5h extrapolation.

(c) Performance counter 0D6h extrapolation.

(d) Performance counter 0D7h extrapolation.

(e) Performance counter 0D8h extrapolation.

(f) STM aborted transaction cycles extrapolation.

(g) Stalled cycles per core.

(h) Scaling factor extrapolation.

(i) Execution time prediction.

Figure 4.4 – `intruder` prediction example.

### 4.3.2 Prediction example

To better explain the prediction process, we use `intruder` from the STAMP benchmark suite as an example. `intruder` is a signature-based network intrusion detection system (NIDS) benchmark that scans network packets and matches them against a set of intrusion signatures. It emulates Design 5 of the NIDS described by Haagdorens et al. in [67]. Packets are processed in parallel and go through three phases: capture, reassembly, and detection. In the version included in STAMP, the capture and reassembly phases are each enclosed in STM transactions.

For this example, we use a machine with four AMD Opteron 6172 processors, each containing two chips with 6 cores each, clocked at 2.1GHz (48 cores in total). ESTIMA uses only one processor of the machine (12 cores) for measurements and targets a machine with four such processors. Hence, the measurements machine is a 12-core machine, while the target machine is a 48-core machine. We then execute the application on the target machine and measure the stalled cycles and execution time for up to 48 cores. We present the extrapolated and measured values in Figure 4.4. The vertical lines in the figures represent the maximum number of cores used for the measurements.

The first step of the process is to collect performance counters for executions of the application when using up to 12 cores (the application is configured to use as many threads as cores available). For this AMD Opteron machine, the performance counters that measure backend stalled cycles are the ones presented in Table 4.2 [12]. `intruder` uses software transactional memory as a concurrency control mechanism. We configure the SwissTM software transactional memory runtime [49] to report aborted transaction cycles for the application and configure ESTIMA to use these cycles.

ESTIMA then approximates each stalled cycle category individually. It creates multiple functions for each category and chooses the function that minimizes the RMSE at the checkpoints. With one function for each stalled cycle category, ESTIMA can extrapolate the stall values for higher core counts. In Figures 4.4a-f we present the result of this process. ESTIMA uses the measurements left of the vertical line and produces the functions presented. In each figure, we also show the measured values on all 48 cores of the *Opteron* machine.

After the performance counter values have been approximated, ESTIMA computes the stalled cycles per core, shown in Figure 4.4g. It is important to note here that although the stalled

Table 4.2 – Hardware performance counters used for the *Opteron* machine.

| Event Code | Event Description |
|---|---|
| 0D2h | Dispatch Stall for Branch Abort to Retire |
| 0D5h | Dispatch Stall for Reorder Buffer Full |
| 0D6h | Dispatch Stall for Reservation Station Full |
| 0D7h | Dispatch Stall for FPU Full |
| 0D8h | Dispatch Stall for LS Full |

cycles for each category increase, the total number of stalled cycles divided by the number of cores decreases for up to 12 cores. This quantity starts increasing for more cores, hinting at a slowdown for the application for higher core counts. Intuitively, as the application runs on more cores, stalled cycles increase *"faster"*. That means that by scaling up the application, we introduce overheads that surpass the benefits in terms of performance. This observation validates the reasons behind ESTIMA's use of fine-grain stalls in the place of an aggregate event (as discussed in Section 2.2.4). By only examining the total number of backend stalls (which is what an aggregate performance counter would report) in Figure 4.4g, we notice that the slowdown of the application is not visible for measurements with up to 12 cores. As a result, if ESTIMA simply extrapolated the values of such an aggregate counter, it would fail to predict the behavior of `intruder` for higher core counts, similarly to the time extrapolation method. In contrast, by using fine-grain stalls, ESTIMA captures the behavior of each stall cycle category and extrapolates them individually. The resulting values, when combined, are able to predict the fast increase in the total number of stalls, resulting in accurate predictions.

The correlation of stalled cycles to execution time involves a scaling factor that connects the two quantities. ESTIMA computes the values of this factor for up to 12 cores using the stalled cycles per core and the execution time values collected. It then approximates this factor. For this approximation, it chooses the function that produces execution time predictions that have the highest correlation to stalled cycles per core. The approximation of the scaling factor is presented in Figure 4.4h. Finally, using this scaling factor, ESTIMA predicts the execution time of the application. We then measure the execution time of `intruder` for up to 48 cores of the machine and use the measurements to evaluate our prediction. Both the predicted and measured execution times are presented in Figure 4.4i. ESTIMA successfully predicts the scalability of the application and the slowdown it exhibits for higher core counts.

## 4.4 Evaluation

### 4.4.1 Implementation

We implement ESTIMA in Python. We integrate the functionality in an easy-to-use tool. For the function approximation, we use the `pythonequation` Python library from [132] in order to create functions based on specific kernels and fit them to collected values of stalled cycles. ESTIMA offers a variety of options for different prediction scenarios. It can either discover the number of cores of the machine it runs on, or take the number of cores to use as an input parameter. ESTIMA discovers the topology of the cores and uses cores within the same socket first. It supports current x86 processor families by both Intel and AMD, and uses all the available events for backend stalls on each architecture. Extending ESTIMA to support additional families of processors is straightforward, by adding the necessary performance counters that need to be collected.

In order to improve the accuracy of predictions, ESTIMA enables the use of *plugin* components. The user can specify additional categories of stalled cycles that can be used for the predictions, either at the hardware, or the software level. ESTIMA takes a configuration file that includes the path to the file the stalls are reported in (including special files like *stdout* or *stderr*), as well as the expression that is used to report the cycles. ESTIMA can apply a function to the collected values (e.g., *min*, *max*, *sum*, *average*) and use the resulting values for its predictions. For example, extending ESTIMA to take into account Intel's RTM [82] would require including additional stalled cycle events (currently there are no available events that measure aborted RTM transaction cycles, but rather only aborted transactions). The way ESTIMA collects additional stalls can vary between applications. In our evaluation, for the collection of synchronization overheads we use a thin wrapper around the `pthread` library. For the applications that use transactional memory, we use SwissTM [49, 50] with detailed statistics enabled, which reports the duration of committed and aborted transactions.

### 4.4.2 Evaluation setup

We evaluate ESTIMA using three different machines. The first is a desktop *Intel Core i7* Haswell machine with 4 cores clocked at 3.4GHz (8 hardware threads in total). The second machine is a 4-processor AMD Opteron 6172 one, with each CPU containing two chips with 6 cores each, clocked at 2.1GHz (48 cores in total). In the remainder of the text we refer to this machine as *Opteron*. We also use a machine that has 2 Intel Xeon E5-2680 v2 processors with 10 cores each, clocked at 2.80GHz (40 threads in total). We refer to this machine as *Xeon20*.

We use several applications to evaluate ESTIMA. These span a variety of workloads, with different lengths of critical sections, levels of contention and synchronization techniques. In total, we use 21 different workloads, among which 8 are STM-based.

### 4.4.3 Extrapolating to different machines

We start our evaluation with two production applications, in a realistic setting. We use `memcached` and a `SQLite` application. We use a desktop machine for our measurements and predict the scalability of the applications on a server machine. We then execute the applications on the server machine and evaluate the accuracy of our predictions.

In our first experiment, we use ESTIMA to predict the scalability of `memcached`. We use ESTIMA on the desktop Haswell machine and target *Xeon20* with our predictions. We run clients on the same machine as the `memcached` server to remove any network effects. The client and dataset are from `cloudsuite` [61], with a scaling factor of 10x. We use the number of workers and connections that produces the highest throughput. The workload is read-mostly and objects have a size of 550 bytes.

ESTIMA collects stalled cycles and execution time from the `memcached` server using up to 3 hardware threads of the desktop machine (clients run on all other hardware contexts) and

Table 4.3 – Hardware performance counters used for the latest *Intel* processors.

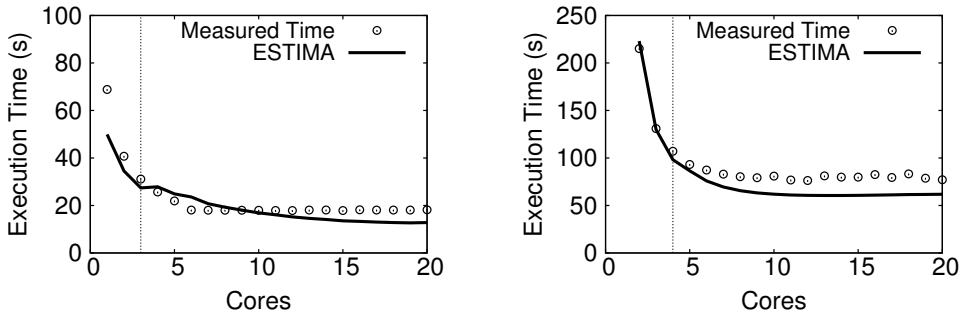
| Event Code | Event Description |
|---|---|
| 0487h | Stalled cycles due to IQ full |
| 01A2h | Cycles allocation stalled due to resource-related reasons |
| 04A2h | No eligible RS entry available |
| 08A2h | No store buffers available |
| 10A2h | Re-order buffer full |

extrapolates its performance to a machine 7 times its size. The performance counters that measure backend stalled cycles for our *Intel* machines are presented in Table 4.3 [82]. As presented in Section 4.3.1, ESTIMA uses measured execution time to correlate stalled cycles to the execution time of the application for higher thread counts. In this experiment, because the machines have processors with different frequencies, the measured execution time is also scaled using the ratio of execution frequencies.

We then measure the execution time of the workload on the full *Xeon20* machine. We execute the `memcached` server on one socket of the machine and the clients on the other socket (20 hardware contexts each). We present ESTIMA's prediction and the measured execution time in Figure 4.5a. ESTIMA successfully predicts the scalability of the application. The absolute errors are below 30% for all core counts. ESTIMA successfully predicts that the server will stop scaling, using only three cores for the measurements, while predicting for up to 7 times more cores.

The second experiment uses the SQLite DBMS, and a TPC-C workload with 10GB of data. We use the same desktop machine and target *Xeon20* again. We use *tmpfs* to avoid IO bottlenecks for logging. ESTIMA collects stalled cycles and execution time from the `SQLite` process for up to 4 cores of the desktop machine and extrapolates its performance to a machine 5 times its size. We measure the same stalled cycles and scale the execution frequencies as before.

We then measure the execution time of the workload on the server machine, using all 20 cores. We keep the threads on the same processor when possible. The result of the prediction




(a) `memcached` execution time prediction.

(b) `SQLite` execution time prediction.

Figure 4.5 – Predictions for `memcached` and `SQLite`.

produced by ESTIMA, as well as the actual time measurements from the server machine are presented in Figure 4.5b. ESTIMA successfully predicts the scalability of the application on the server machine. The execution time errors are below 26% for all core counts. ESTIMA successfully predicts that the server will stop scaling, as well as the number of cores for which this will happen. It does so using only four cores for the measurements and predicting for a machine with 5 times more cores.

### 4.4.4 Scaling-up applications

In our previous experiments, we show how we use ESTIMA to extrapolate the scalability of production applications. We now evaluate the extent to which our tool can predict the scalability of applications by using 3 suites of in-memory benchmarks: STAMP [115], Parsec [22] and standard data structure micro-benchmarks (used in [50]). We also use a modified k-nearest neighbors (*KNN*) calculation kernel, commonly used in recommender systems. The bench-

Table 4.4 – Maximum prediction errors with measurements on 1 processor of each machine (12 cores for *Opteron* and 10 cores for *Xeon20*).

| | *Opteron* Errors (%) | | | *Xeon20* Errors (%) |
|---|---|---|---|---|
| Benchmark | 2 CPUs | 3 CPUs | 4 CPUs | 2 CPUs |
| lock-based HT | 7.8 | 8.3 | 8.9 | 41.7 |
| lock-based SL | 27.7 | 24.3 | 21.4 | 16.1 |
| lock-free HT | 3.3 | 3.4 | 3.7 | 15.8 |
| lock-free SL | 13.2 | 10.4 | 9.9 | 24.8 |
| **stamp:** | | | | |
| genome | 4.4 | 4.4 | 4.6 | 6.3 |
| intruder | 9.2 | 22.1 | 31.9 | 30.0 |
| kmeans | 50.3 | 50.9 | 17.0 | 30.2 |
| labyrinth | 15.4 | 15.0 | 18.4 | 9.9 |
| ssca2 | 2.8 | 4.6 | 8.1 | 21.4 |
| vacation-high | 14.7 | 14.3 | 10.3 | 16.8 |
| vacation-low | 18.9 | 18.5 | 25.0 | 10.0 |
| yada | 8.1 | 23.0 | 15.1 | 40.3 |
| **parsec:** | | | | |
| blackscholes | 3.7 | 4.4 | 2.9 | 13.9 |
| bodytrack | 1.3 | 3.0 | 5.9 | 8.5 |
| canneal | 10.7 | 12.4 | 8.3 | 6.4 |
| raytrace | 2.7 | 3.6 | 4.6 | 1.7 |
| streamcluster | 15.6 | 59.0 | 88.8 | 20.1 |
| swaptions | 10.6 | 14.7 | 20.3 | 9.3 |
| K-NN | 11.5 | 22.5 | 32.0 | 13.1 |
| Average | 11.3 | 16.8 | 17.7 | 17.7 |
| Std. Dev. | 11.2 | 15.0 | 18.9 | 11.0 |
| Max. | 50.3 | 59.0 | 88.8 | 41.7 |

mark suites we use are written in C/C++ and compiled using GCC, while the KNN calculation kernel is written in Java and compiled using GCJ. We use one CPU of the *Opteron* and *Xeon20* machines for our measurements (12 and 10 cores respectively) and then predict for up to 4 and 2 CPUs respectively (the full machines).

In Table 4.4, we present the summary of the predictions produced by ESTIMA for *Opteron* and *Xeon20*. The errors presented are the maximum errors observed when using one processor and targeting up to the full machines with our predictions (2, 3 and 4 processors of *Opteron* and 2 processors of *Xeon20*). For brevity, we present results using *Opteron*, for which we have predictions for higher core counts. In our examples, we also present the results of the time extrapolation method presented in Section 2.2.4 (*time extrapolation*). This method involves directly extrapolating time measurements of an application, using the function kernels that ESTIMA uses to extrapolate stalled cycle measurements. This approach is similar to using aggregate events for our measurements. It fails to predict changes in the application behavior that are not evident from the measurements, which explains the significant differences in accuracy when compared to ESTIMA. We highlight the biggest of these differences in accuracy between *time extrapolation* and ESTIMA in Figure 4.6.

Our benchmark evaluation shows that ESTIMA is successful in predicting the scalability of most benchmarks with small prediction errors. Out of 19 workloads used for the evaluation of ESTIMA:

- ESTIMA is successful in predicting the scalability of the applications used. There are no

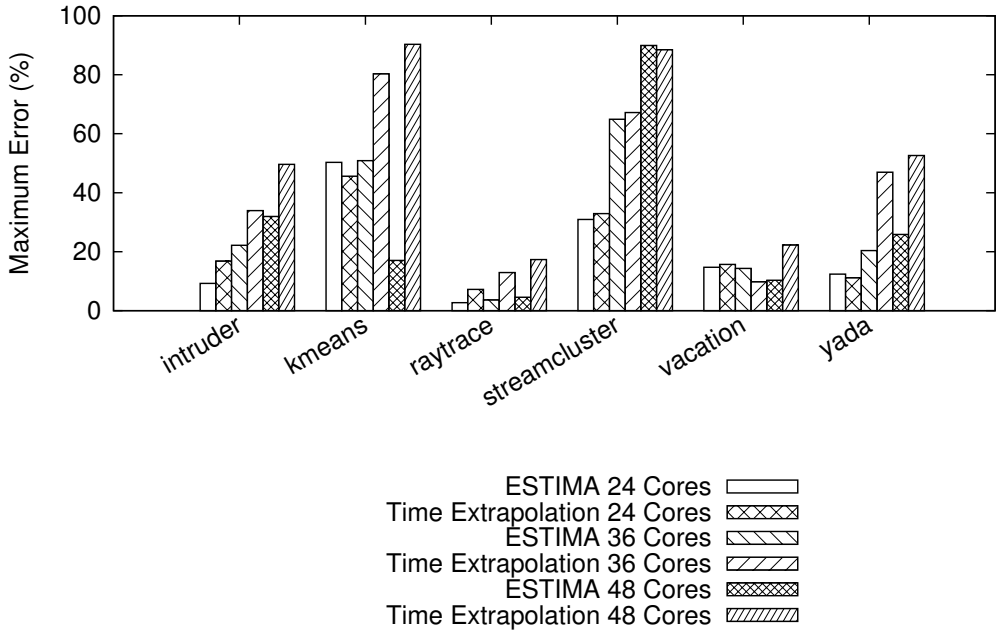


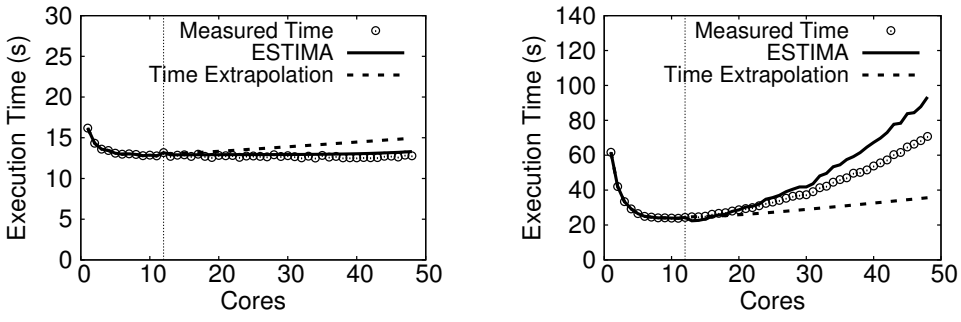

Figure 4.6 – Comparison of errors between ESTIMA and time extrapolation.

cases where ESTIMA incorrectly predicts that an application will or will not scale. ESTIMA also successfully predicts the core counts for which an application will stop scaling,

- When extrapolating from one socket of *Xeon20* to the full machine (double the number of cores used for the measurements), 15 workloads have execution time prediction errors lower than 25% and 9 of them have errors lower than 10%.
- When extrapolating from one socket of *Opteron* to the full machine (four times the number of cores used for the measurements), 16 workloads have execution time prediction errors lower than 25% and 9 of them have errors lower than 10%.

In Figure 4.7a we present an example of a prediction result, using `raytrace` from the PARSEC benchmark suite. `raytrace` is an Intel RMS application which uses a version of the raytracing method that would typically be employed for real-time animations such as computer games, optimized for speed rather than realism. ESTIMA accurately predicts its scalability, with the maximum execution time prediction error observed for predictions for up to 48 cores being 4.6%. This is in contrast to the time extrapolation method, which produces errors up to 17.3%.

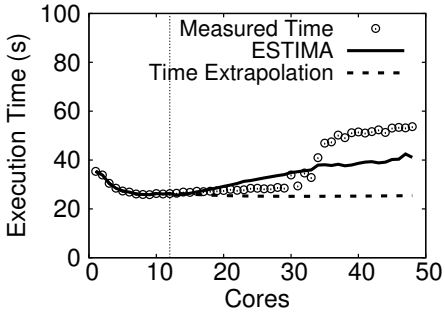
The main advantages of ESTIMA appear when predicting changes in the behavior of an application, such as the ones seen in `intruder` and `yada` from the STAMP benchmark suite, presented in Figures 4.7b and 4.7c. `intruder` has already been introduced in Section 4.3.2. `yada` implements Ruppert's algorithm for Delaunay mesh refinement [141]. The input consists of an initial mesh and threads identify the triangles of the mesh for which the minimum angle



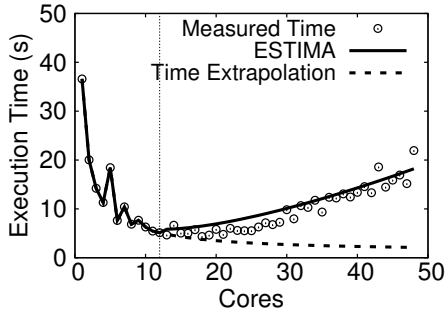

(a) `raytrace` execution time prediction.




(b) `intruder` execution time prediction.




(c) `yada` execution time prediction.




(d) `kmeans` execution time prediction.

Figure 4.7 – Predictions using ESTIMA.

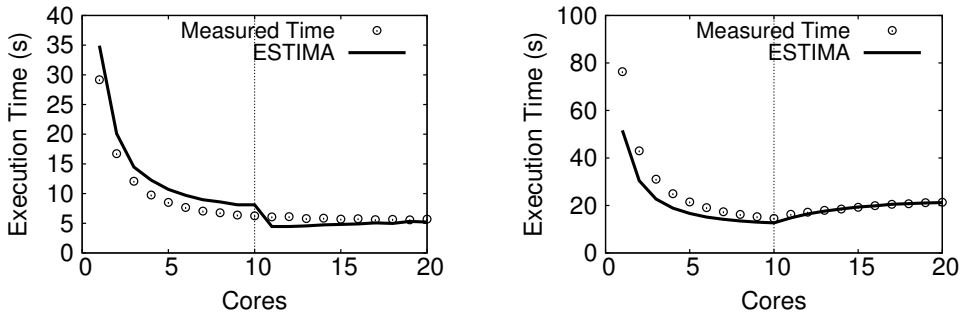
is below some threshold. Once such triangles are found, new points are added to the mesh and the process continues with new triangulations. For both workloads ESTIMA successfully predicts the changes that appear, as well as the scalability of the applications. These cases demonstrate the advantage of using stalled cycles for our predictions, as the trends in stalled cycles appear before their effect in performance is significant enough. This is unlike time extrapolation, which fails to predict the scalability trends and has higher prediction errors (up to 81% and 130% higher for `intruder` and `yada` respectively).

Another interesting example is `kmeans` from the STAMP suite. `kmeans` is a partition-based clustering benchmark that represents a cluster by the mean value of all objects contained in it. We present a scalability prediction for `kmeans` on the *Opteron* machine in Figure 4.7d. Although the maximum prediction error for `kmeans` is 50.9% in Table 4.4, the prediction is accurate. The high error value is the result of fluctuations in `kmeans`' execution time for different core counts, which the prediction does not follow. Nevertheless, ESTIMA successfully predicts the scalability of the application. As with `intruder` and `yada`, the scalability degradation of `kmeans` is not evident in the performance on the measurement machine. However, ESTIMA successfully captures the trends in stalled cycles and predicts the performance degradation accurately.

### 4.4.5 Weak scaling

Using a larger machine enables users to also use bigger datasets, due to the bigger amount of memory typically available. To evaluate how we can use ESTIMA with changing workload sizes, we use measurements of an application on a machine and predict its scalability on a machine with double the number of cores, but also with twice as large a dataset. We use `genome` and `intruder` from the STAMP benchmark suite, introduced earlier in this section. We run our experiments on the *Xeon20* machine. We use both applications with the default datasets from the STAMP suite.

ESTIMA executes measurements on one processor of the *Xeon20* machine and targets the full machine. We configure ESTIMA with a target dataset size of 2x. ESTIMA uses the same




(a) `genome` execution time prediction.

(b) `intruder` execution time prediction.

Figure 4.8 – Predictions with changing workload sizes.

technique for extrapolating the scalability of the applications, but also measures the memory footprint of the execution. During the extrapolation process, ESTIMA scales the extrapolated values accordingly, in order to produce predictions for the target machine and dataset, for both applications.

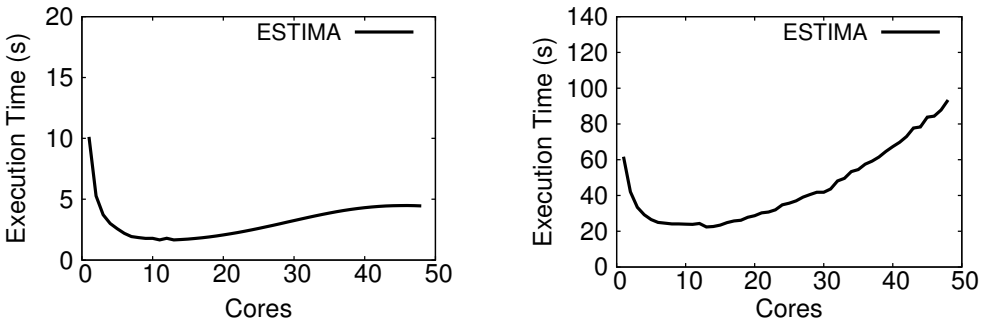
We then execute the applications on the full *Xeon20* using the bigger dataset. We present both ESTIMA's prediction, as well as the measured execution time in Figure 4.8. ESTIMA successfully predicts the scalability for both applications. The predictions are accurate in absolute terms too. The maximum errors (excluding single core performance) are 28% for `intruder` and 29% for `genome`. The most significant errors appear for single core performance of `intruder` on the bigger machine. This can be explained due to the simple scaling that we use in order to target the bigger dataset, which does not accurately connect the performance on a single core.

Although with higher errors than their strong scaling counterparts, these predictions show how, with a simple technique, ESTIMA can scale both the workload size and together the core counts at the same time, while maintaining the generality and wide applicability of ESTIMA, that is without using complex models for the interaction between the workload and the different cache levels. We expect more complex methods, such as scaling different stalled cycle categories differently based on measurements, to be able to produce even more accurate predictions for different workload sizes.

### 4.4.6 Identifying future bottlenecks

In our evaluation so far, we present how ESTIMA can be used to extrapolate stalled cycles in order to predict the scalability of an application. A question that arises is the following: *can* ESTIMA *help developers identify the sources of poor scalability in their applications, before such poor scalability even appears?* We now show how we can use ESTIMA to identify bottlenecks in two examples. We use two applications that use different concurrency control mechanisms: `streamcluster` and `intruder`. For both applications ESTIMA collects stalled cycles from both hardware and software sources. For `streamcluster`, we create a thin wrapper around the pthread library calls. For `intruder`, we simply configure the SwissTM runtime library to report aborted transaction cycles.

ESTIMA extrapolates the scalability of the two applications on *Opteron*. It uses measurements on one processor of the machine (12 threads) and extrapolates to all four processors (48 threads). We show the results of these extrapolations in Figure 4.9. Both applications exhibit slowdown for high core counts. We observe the individual stalled cycle category extrapolations and identify the ones that contribute most to stalls for higher core counts. We then use the *perf* linux tool to pinpoint the most significant sources of the reported stalls in our measurements. For `streamcluster`, we notice a high number of stalled cycles in the *pthread_mutex_trylock* function of the PARSEC barriers. This leads us to identify the mutexes used as a potential scalability bottleneck. For `intruder`, we similarly notice a high number of stalled cycles from aborted STM transactions in the *processPackets* function, and more specifically in the



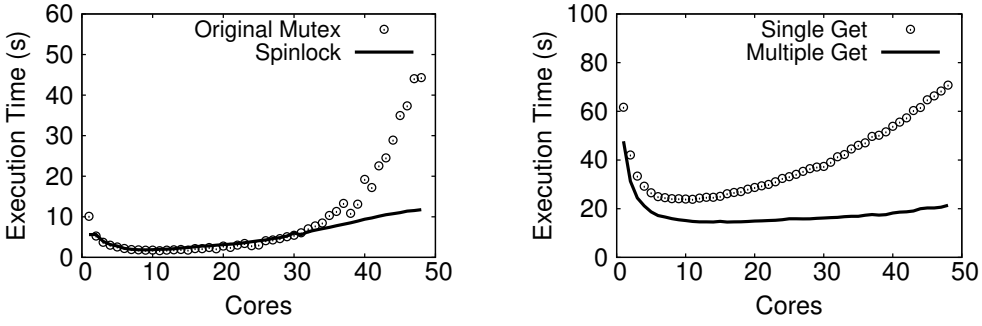

(a) Prediction for `streamcluster`.

(b) Prediction for `intruder`.

Figure 4.9 – Predictions for `streamcluster` and `intruder` using `ESTIMA`.

*TMDECODER_PROCESS* call. By examining the function code, we understand that aborted transactions are the result of contention for a shared data structure.

In order to fix the problems identified, we modify the two applications. For `streamcluster`, we replace the standard pthread mutexes used by PARSEC with test-and-set spinlocks, which incur lower synchronization overheads. For `intruder`, we modify the application to decode more elements in every step. The measurements on the full *Opteron* machine validate our findings for the scalability bottlenecks. We show the original and modified applications' performance in Figure 4.10. For `streamcluster`, we improve its execution time by up to 74%. Similarly, our optimization improves `intruder`'s performance by up to 70%. Evidently, the applications continue to scale poorly. There do exist more bottlenecks that we could identify and try to remove. Nevertheless, the goal of this example is to showcase how ESTIMA can be used to identify future scalability bottlenecks. It is important to note here that identifying bottlenecks is not the main goal of ESTIMA, and as such, it cannot replace tools specifically designed for this purpose.




(a) `streamcluster` scalability.

(b) `intruder` scalability.

Figure 4.10 – Improving the scalability of `streamcluster` and `intruder` using `ESTIMA`'s predictions.

## 4.5 Discussion

### 4.5.1 Stalled cycles and scalability

The main insight behind ESTIMA is the usage of backend stalls for scalability extrapolations. This assumes that stalled cycles contain the necessary information about the scalability of an application. Our evaluation shows that indeed, by using backend stalls from both hardware and software, ESTIMA successfully extrapolates the scalability of a wide range of applications.

We notice that for some workloads and platforms, prediction errors are higher. Although these cases are limited, it is crucial to understand the cause of such errors. In order to do so, we need to first consider *where* errors can occur in ESTIMA. The two main components of the extrapolations ESTIMA performs are the following:

1. The extrapolation of stalled cycles from hardware (and optionally software).

Table 4.5 – Correlation of stalled cycles per core with execution time for the full machines.

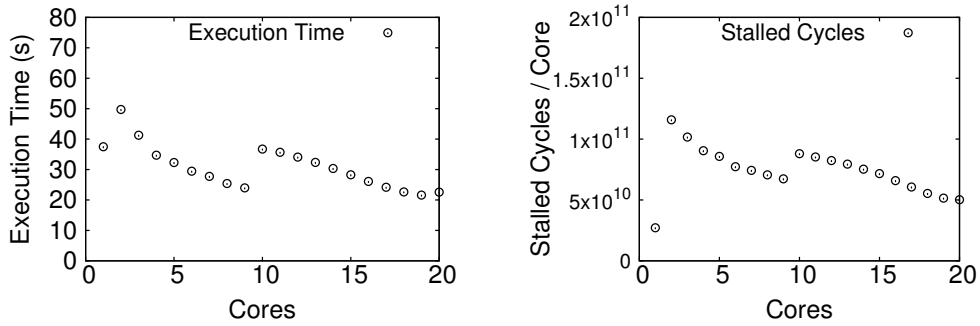
| Benchmark | *Opteron* | *Xeon20* | *Xeon48* |
|---|---|---|---|
| lock-based HT | 0.71 | 0.66 | 0.93 |
| lock-based SL | 1.00 | 1.00 | 1.00 |
| lock-free HT | 1.00 | 1.00 | 1.00 |
| lock-free SL | 0.83 | 0.81 | 0.70 |
| **stamp:** | | | |
| genome | 1.00 | 1.00 | 1.00 |
| intruder | 1.00 | 0.97 | 0.92 |
| kmeans | 0.88 | 0.98 | 0.96 |
| labyrinth | 0.99 | 1.00 | 1.00 |
| ssca2 | 0.99 | 1.00 | 1.00 |
| vacation-high | 1.00 | 1.00 | 0.99 |
| vacation-low | 0.99 | 1.00 | 0.98 |
| yada | 0.62 | 0.95 | 0.77 |
| **parsec:** | | | |
| blackscholes | 1.00 | 1.00 | 1.00 |
| bodytrack | 1.00 | 1.00 | 1.00 |
| canneal | 0.97 | 0.99 | 0.95 |
| raytrace | 0.94 | 0.98 | 0.96 |
| streamcluster | 0.84 | 1.00 | 0.81 |
| swaptions | 1.00 | 1.00 | 1.00 |
| K-NN | 1.00 | 1.00 | 0.99 |
| Average | 0.93 | 0.97 | 0.94 |
| Std. Dev. | 0.11 | 0.08 | 0.09 |
| Min. | 0.62 | 0.66 | 0.70 |

2. The correlation of the extrapolated stalled cycles to execution time, and the extrapolation of the scaling factor that connect the two quantities.

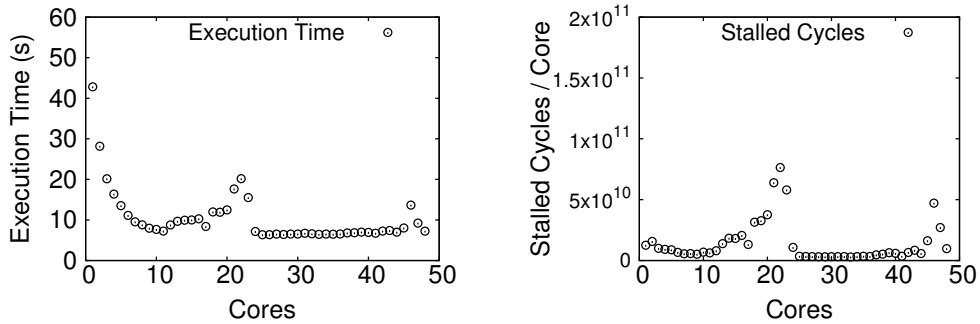
Both are potential sources of errors. Function approximation can produce functions that do not extrapolate accurately a category of stalled cycles or the scaling factor that connects stalled cycles to execution time. In this last step, inaccuracies can result in significant errors in the prediction. Nevertheless, ESTIMA assumes that stalled cycles can tell the full story of scalability, and errors are due to the limitations of function approximation. If this does not hold true, extrapolation errors can be significantly higher. Even worse, ESTIMA could predict that an application will continue scaling, while in fact it will not.

To evaluate this assumption, we conduct an experiment using an Intel machine with 4 E7-4830 v3 processors (48 cores – *Xeon48*). We execute all benchmarks for all core counts and measure both stalled cycles and execution time. We then calculate the correlation between stalled cycles per core and execution time. Ideally this correlation should be 1.0, so that errors are mainly due to function approximation. More importantly, such a high correlation would indicate that stalled cycles follow the changes of execution time and will correctly predict whether an application will stop scaling, as well as the core count for which this will happen.

Correlations across applications and machines (Table 4.5) are higher than 0.95 for the vast majority of cases, supporting ESTIMA's use of stalled cycles. What this implies is that errors in




(a) `lock-based hash table` on *Xeon20*.




(b) `lock-free skip list` on *Xeon48*.

Figure 4.11 – Execution time and stalled cycles for two data structure microbenchmarks.

predictions are mainly due to function approximation errors. These numbers include software stalls for the applications of the STAMP benchmark suite, as well as `streamcluster` from the PARSEC suite. In the presented results, we notice cases with lower correlation, namely the lock-based hash table microbenchmark on the *Xeon20* machine and the lock-free skip list on the *Xeon48* machine. We present the execution time and stalled cycle measurements for both workloads in Figure 4.11. Execution time and stalled cycles per core have similar curves. The correlation is lower because of small changes in core to core measurements that are not in sync for the two curves. However, in both cases ESTIMA accurately extrapolates scalability, as we show in our evaluation (Table 4.4).

### 4.5.2 Backend vs. frontend stalled cycles

When designing ESTIMA, we opted to use backend stalled cycles as our main tool. As discussed in Section 4.2, there also exist frontend stalls, which mainly refer to instruction fetch stalls (e.g., misses in the instruction cache). We chose to not use these stalls for two reasons: (a) bottlenecks that hinder scalability do not typically stem from instruction fetching, but rather the execution, and (b) there is a limit to the number of hardware counters that modern platforms can accurately monitor at the same time without imposing additional overheads to an application.

In order to verify the extent to which frontend stalls indeed do not contribute to the accuracy of ESTIMA, we extend the experiment of the previous subsection. We measure both frontend and backend stalls and calculate the correlation between the number of stalled cycles (including frontend stalls) per core and the execution time of a workload. We present the differences in correlation to execution time between with and without including frontend stalled cycles in Table 4.6. Positive values in the table signify that using frontend stalls improves the correlation, while negative values signify that using frontend stalled cycles degrades the correlation between the two quantities.

As seen from the table, the average improvement in the results is either close to zero, or negative. This indicates that frontend stalls do not contribute additional information about the scalability of the applications. In some cases using frontend cycles can decrease correlation to execution time by more than 10%, which can result in significant prediction errors for ESTIMA. These findings confirm our decision to not include frontend cycles in ESTIMA.

Similarly to frontend stalls, we could omit stalls that have an insignificant effect. Indeed, not all backend stall categories contribute equally to the total. However, the most important stalled cycle category varies per application. For example, while the *0D7h* event (Table 4.2) contributes less than 0.01% of the total number of stalls for `vacation-high` and less than 0.1% for `intruder` on *Opteron*, it contributes more than 30% of the total stalls for `blackscholes` on the same machine. As such, ESTIMA uses all the available backend stall events of a processor.

Table 4.6 – Frontend+backend stalled cycles improvement over backend-only stalls (%).

| Benchmark | *Opteron* | *Xeon20* | *Xeon48* |
|---|---|---|---|
| lock-based HT | 2.09 | -6.93 | -2.63 |
| lock-based SL | 0.00 | -0.52 | 0.00 |
| lock-free HT | 0.01 | 0.00 | -0.16 |
| lock-free SL | 7.52 | 6.76 | 3.53 |
| **stamp:** | | | |
| genome | 0.02 | 0.02 | -0.10 |
| intruder | 0.08 | 1.31 | -9.98 |
| kmeans | 2.86 | -12.07 | 5.67 |
| labyrinth | 0.37 | -0.28 | -0.02 |
| ssca2 | -0.71 | -0.05 | -0.46 |
| vacation-high | 0.00 | -0.02 | -0.03 |
| vacation-low | -0.04 | -0.01 | -0.10 |
| yada | 3.43 | -0.10 | 5.57 |
| **parsec:** | | | |
| blackscholes | 0.00 | 0.00 | -0.01 |
| bodytrack | 0.00 | -0.01 | -0.04 |
| canneal | -0.03 | 0.01 | -0.11 |
| raytrace | 0.13 | -0.03 | 0.13 |
| streamcluster | 0.80 | -14.79 | -2.67 |
| swaptions | 0.00 | 0.00 | 0.00 |
| K-NN | 0.05 | 0.52 | -0.12 |
| Average | 0.87 | -1.38 | -0.08 |
| Std. Dev. | 1.89 | 4.73 | 3.16 |
| Max. | 7.52 | 6.76 | 5.67 |
| Min. | -0.71 | -14.79 | -9.98 |

### 4.5.3 Software stalled cycles

ESTIMA by default uses only hardware stalls for its extrapolations. However, as presented in Section 4.4.1, users can configure ESTIMA via plugins to also include software stalls in its extrapolations. This way the accuracy of its predictions can be improved.

We measure software stalled cycles for all applications from the STAMP suite, by configuring the STM runtime to report aborted transaction cycles. Because of the nature of STM, the effect of software stalls is expected to be high: contention for resources and data memory locations leads to aborted transactions. Thus, although at the hardware level instructions are executed, all the work is discarded when a transaction is aborted, not contributing to useful work at the application level.

We additionally measure software stalled cycles due to synchronization for applications that use the `pthread` library. We do so by writing a thin wrapper around the library that measures

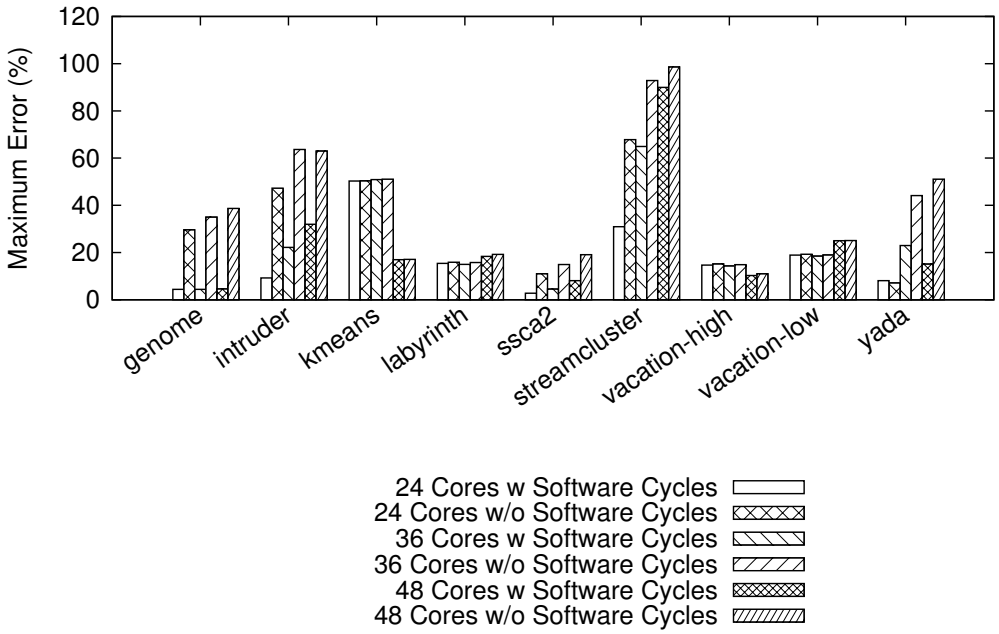
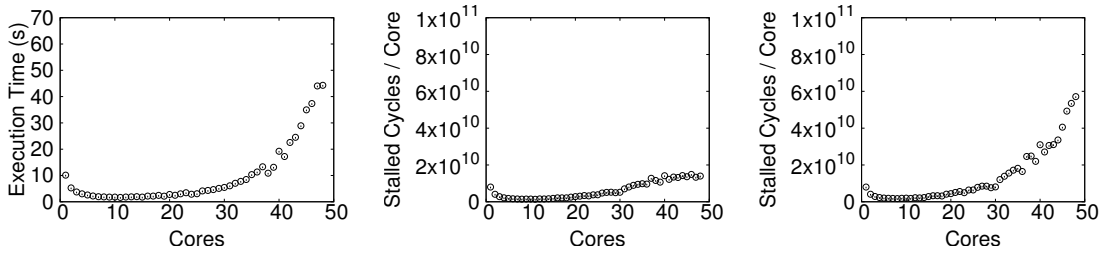



Figure 4.12 – Comparison of prediction errors with and without software stalled cycles.

the cycles each threads spends spinning on locks and barriers. We measure these synchronization cycles for `streamcluster` from the PARSEC suite, as well as for `genome` and `ssca2` from the STAMP suite.

In Figure 4.12 we present all the applications for which we configure ESTIMA to additionally use software stalled cycles. We show the prediction errors for the applications with and without software cycles. We observe that using software cycles can significantly improve the prediction accuracy: by 57% on average, and for certain applications (e.g., `genome`) by up to 87% when targeting a machine with four times the number of cores.

An application for which hardware stalls alone do not accurately capture its scalability is `streamcluster` from the PARSEC benchmark suite. For `streamcluster`, synchronization overheads introduce a significant bottleneck (one that we identify using ESTIMA in Section 4.4.6). The execution time of `streamcluster` on our *Opteron* machine can be seen in Figure 4.13a. When using only hardware stalls (Figure 4.13b), stalled cycles per core do not show the synchronization bottleneck, resulting in lower correlation with execution time (0.86). When incorporating the cycles spent on synchronization, stalled cycles per core give a more complete image of the scalability of the application (Figure 4.13c), resulting in very high correlation to execution time (0.98).




(a) Execution time.

(b) Stalled cycles per core without synchronization cycles.

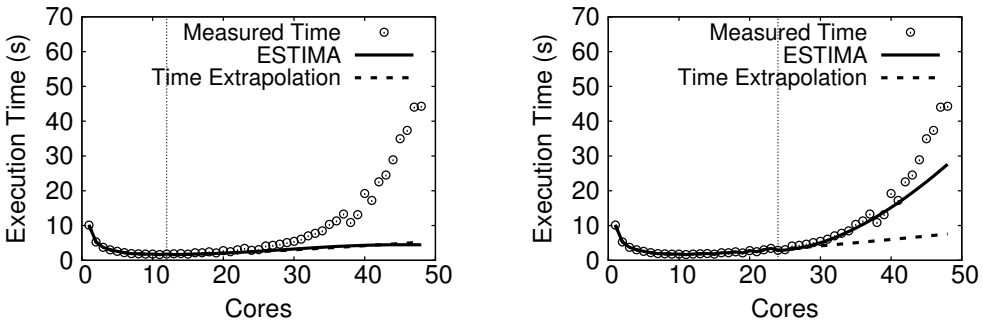
(c) Stalled cycles per core with synchronization cycles.

Figure 4.13 – Effect of software stalled cycles for `streamcluster`.

### 4.5.4 Limitations

Table 4.4 shows that predictions for `streamcluster` from the PARSEC benchmark suite exhibit high absolute prediction errors. `streamcluster` is a clustering benchmark, which, for a stream of input points, finds a predetermined number of medians, so that each point is assigned to its nearest center. We show both the extrapolated and measured execution time of the application in Figure 4.14a.

The behavior of `streamcluster` changes significantly for more than 30 cores. ESTIMA successfully captures the slowdown of the application, but with higher absolute errors, because there is no hint of this performance change in the measured stalls. When using 24 cores for the measurement (2 sockets of *Opteron*), the prediction is significantly better, as seen in Figure 4.14b. This shows the main limitation of ESTIMA. Although stalled cycles show trends before they have an impact on performance, as discussed in Section 4.2, there are cases where significant changes in the application happen for higher core counts. In this example, the synchronization overheads, together with memory bandwidth saturation cause slowdown for core counts greater than 36. This behavior is not captured by stalled cycles when using measurements for up to 12 cores.




(a) `streamcluster` execution time prediction.

(b) `streamcluster` execution time prediction with measurements for up to 24 cores.

Figure 4.14 – Predictions for `streamcluster`.

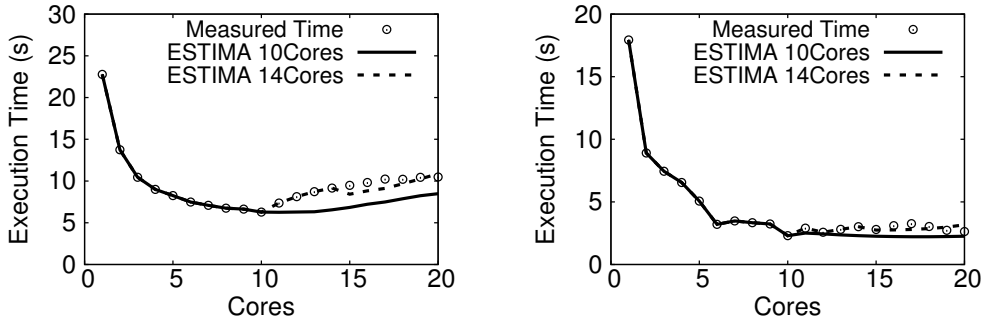
Moreover, ESTIMA is not a silver bullet for predicting the scalability of parallel applications. ESTIMA's main use case involves predictions for machines with similar architectures. ESTIMA successfully predicts the performance of an application across such machines, as we show in our evaluation. However, cross-platform predictions with significant differences between the measurements and target architectures (e.g., using measurements from an Intel machine to predict performance on a SPARC machine), will typically result in higher errors. Similarly, ESTIMA is not meant to predict the performance of an application for very different workload configurations, as it does not rely on static or dynamic analysis of the target application. We believe that these shortcomings are outweighed by the simplicity and generality of ESTIMA.

### 4.5.5 NUMA effects

Because of its nature, ESTIMA does not capture effects that are not present in the measurements machine. As an application scales to more sockets, effectively changing the underlying architecture (e.g., introducing NUMA effects), prediction accuracy will be lower. The effect of NUMA accesses on the scalability of an application varies with the application, the target machine and the dataset. Capturing the interaction of such factors would require a detailed model of the application (e.g., memory access patterns), which is against ESTIMA's goal of wide applicability and ease of use.

However, this is not the case on our *Opteron* machine. The reason lies in its architecture, which enables ESTIMA to account for NUMA effects. Each processor of *Opteron* consists of two chips, each containing 6 cores (12 in total per processor). Thus, memory accesses on one processor introduce remote accesses between the two chips. With these effects present in measurements, ESTIMA accurately extrapolates their impact, resulting in predictions with high accuracy.

When observing the results for *Xeon20*, we see the average prediction errors are higher. Contrary to *Opteron*, *Xeon20* is a typical NUMA machine with two sockets. As such, single-socket execution measurements do not include the effects of NUMA accesses. Thus, ESTIMA fails to




(a) `intruder` execution time prediction.

(b) `kmeans` execution time prediction.

Figure 4.15 – Predictions with NUMA effects captured.

capture the trends that they cause, resulting in higher prediction errors for high core counts. Including these effects significantly improves the accuracy of predictions. For example, using measurements with more than 10 cores involves cores from the second socket, capturing non-local accesses and improving the prediction accuracy. We present two examples of such extrapolations in Figure 4.15.

To further demonstrate how non-uniformity can be accounted for in measurements, we conduct an experiment with our two Intel platforms. We use ESTIMA to execute and measure stalled cycles for all of the benchmarks used in our evaluation on both sockets of *Xeon20*, targeting the *Xeon48* machine introduced in Section 4.5.1 (2.5x the number of cores). We then execute the application on the *Xeon48* machine and measure the execution time for the same workload. Table 4.7 presents the maximum prediction errors for each application. In the same table we have also copied the errors observed when extrapolating the scalability of one socket of *Xeon20* to the full machine (from Table 4.4). The results clearly show that

Table 4.7 – Maximum prediction errors for predictions targeting our *Xeon48* machine (*Xeon20* prediction errors from Table 4.4 for comparison).

| Benchmark | *Xeon20* Errors (%) | *Xeon20* to *Xeon48* Errors (%) |
|---|---|---|
| lock-based HT | 41.7 | 19.8 |
| lock-based SL | 16.1 | 18.4 |
| lock-free HT | 15.8 | 6.8 |
| lock-free SL | 24.8 | 23.1 |
| **stamp:** | | |
| genome | 6.3 | 8.2 |
| intruder | 30.0 | 5.2 |
| kmeans | 30.2 | 30.0 |
| labyrinth | 9.9 | 18.0 |
| ssca2 | 21.4 | 15.3 |
| vacation-high | 16.8 | 12.8 |
| vacation-low | 10.0 | 10.9 |
| yada | 40.3 | 15.2 |
| **parsec:** | | |
| blackscholes | 13.9 | 13.4 |
| bodytrack | 8.5 | 11.4 |
| canneal | 6.4 | 8.9 |
| raytrace | 1.7 | 4.0 |
| streamcluster | 20.1 | 21.1 |
| swaptions | 9.3 | 11.6 |
| K-NN | 13.1 | 9.2 |
| Average | 17.7 | 13.9 |
| Std. Dev. | 11.0 | 6.5 |
| Max. | 41.7 | 30.0 |

prediction accuracy is significantly improved. The average prediction error falls from 17.7% to 13.9% (an improvement of 21.47%). Additionally, and more importantly, the prediction errors have a lower standard deviation (6.5 instead of 11). What this implies is that errors are better clustered around the average, rather than having multiple very small values, and a few very large ones. This is also evident in the absence of significant errors in the results: the maximum error observed is 30%, down from 41.7%, (an improvement of 28.06%).

## 4.6 Conclusion

In this chapter we introduced ESTIMA, a practical tool for extrapolating the scalability of in-memory parallel applications. ESTIMA is designed to help developers and users predict the scalability of applications with minimum effort, without the need for detailed models. To achieve that, ESTIMA uses stalled cycle measurements at the hardware and optionally at the software level, and function approximation on the collected values. ESTIMA is general and easy to use. It can be applied to *any* in-memory parallel application with minimum effort. It can also take advantage of application-specific user input to further improve the accuracy of its predictions.

ESTIMA produces accurate predictions, as conveyed by our extensive evaluation. We successfully used ESTIMA to predict the scalability of production applications, as well as a wide range of benchmarks. The errors when predicting for a machine up to four times larger than the one available for measurements were lower than 15% for more than half of the applications. Moreover, ESTIMA successfully captured the scalability behavior of all the applications used. Finally, we used ESTIMA to identify scalability bottlenecks in two applications, showing how developers can benefit from ESTIMA during the development phase.

# Part III

# Improving the Scalability of Shared-Memory Applications

# 5 Locking Made Easy

## 5.1 Introduction

*Locking* is arguably the most widely-used synchronization technique in concurrent programming. Essentially, every modern system makes use of locks. Most operating systems (e.g., the Linux kernel), DBMSs (e.g., MySQL), and key-value stores (e.g., Memcached) heavily rely on locks. The wide adoption of locking can be mainly attributed to the need for a simple and fast technique for synchronization. Indeed, locking seems simple at a first glance. However, in practice, there is more to using locks efficiently in systems than meets the eye.

Typically, the programmer must (i) map data to locks, (ii) declare locks, (iii) allocate and initialize locks, (iv) use the locks (i.e., *lock* and *unlock*), and (v) destroy and deallocate the locks. Of course, she has to also select which lock algorithm(s) to use.

These steps are inflexible and error-prone. Every lock object must be explicitly declared, hence, changing the mapping of data to locks can be cumbersome. For example, moving away from the global lock in the Linux kernel required significant effort [21, 104]. Common mistakes during lock-based programming include [33]: (i) accessing uninitialized locks, (ii) trying to acquire the same lock twice, (iii) releasing an already free lock, (iv) releasing a lock that belongs to another thread, and, of course, (v) deadlocks. Theses issues can be difficult to debug.

To make things even more complicated, in order to achieve good performance, one has to fine-tune the locking strategies in use. Indeed, there is no one-size-fits-all lock algorithm [40, 66]. Consider Figure 5.1 that depicts the performance of different locking strategies under increasing contention. Simple spinlocks are very fast on low contention but do not scale well. Queue-based locks scale well, but are slower than spinlocks on low contention and suffer on multiprogrammed environments (i.e., when the number of threads is larger than the number of hardware contexts). Blocking locks (e.g., `pthread` mutexes) are suitable on that latter case, but are very slow on non-multiprogrammed environments.

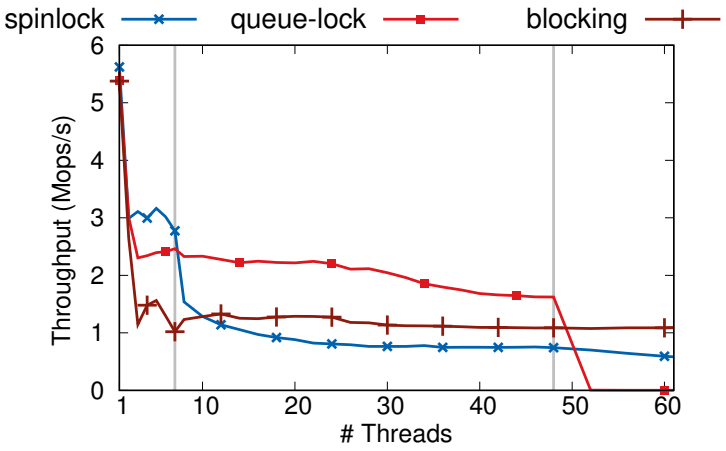



Figure 5.1 – Different lock strategies under varying contention.

Consequently, selecting the "wrong" algorithm can have detrimental performance results. For example, recent work [40, 59, 65, 66, 110] has shown significant performance improvements in middleware software, such as Memcached and LevelDB, by modifying locking. Accordingly, the designer must select the appropriate locking technique *for each* lock object in order to optimize her system as much as possible. However, the "correct" lock algorithm strongly depends on the lock contention levels which, in turn, depend on the workload. Additionally, workloads tend to change over time, thus the correct lock algorithm is a function of time. Ideally, we would like a single lock algorithm that can dynamically adapt to the workload, delivering the best performance among spinlocks, queue-based locks, and blocking locks, at any point in time.

In this chapter, we question whether we can have the cake and eat it too. We present GLS, a *generic locking service* designed to solve all the aforementioned intricacies related to lock-based programming. GLS is a middleware that provides the traditional lock interface, with two main functions to acquire and release a lock. However, in contrast to traditional lock libraries, the locks are fully controlled by GLS, thus the developer does not need to select a lock algorithm, nor to declare, allocate, or initialize any locks. In fact, any arbitrary memory address can be used as a parameter to `gls_lock()`. GLS takes care of mapping the input address to a lock object.

Under the hood, GLS uses a fast concurrent hash table for mapping addresses to objects. As most locks are repeatedly used, this hash table converges to a read-mostly hash table, thus incurring low overhead. Having a central data structure, where all locks are kept, allows us to develop very useful debugging extensions on top of GLS. GLS can detect accesses to uninitialized locks, double locking, etc. Additionally, in §5.3.2, we show how to build low-overhead deadlock detection on top of GLS, as well as lock profiling tools.

More importantly, the programmer does not need to select the lock algorithm for each individual lock; GLS automatically adapts the lock algorithm to the workload. GLS comes with an

adaptive lock called GLK, standing for *generic lock*. GLK keeps track of the contention levels in order to dynamically adapt the algorithm to the needs of the workload. On low contention, GLK behaves as a simple spinlock (i.e., ticket lock [111]). On high contention, GLK turns into a scalable queue-based lock (i.e., MCS lock [111]). On high system load (multiprogramming), GLK transforms to a blocking lock (i.e., mutex lock). The adaptiveness of GLK is per lock, thus a system might contain various locks that operate on different modes depending on their contention levels (e.g., MySQL in §5.4.2). Naturally, in a system with locking already in place, GLK can be used with or without GLS to minimize the overhead.

Additionally, GLS offers explicit interfaces for many state-of-the-art lock algorithms: test-and-set, test-and-test-and-set, ticket, MCS, CLH [37], and mutex, and allows for easy deployment of more algorithms. These interfaces can be used to manually specify the lock algorithm to be employed for a specific lock object. In §5.4.1, we show how we use this interface to re-implement locking in Memcached from scratch. The resulting implementation contains 26% less lock-related code than the initial design and delivers 14% higher throughput on our benchmarks.

Table 5.1 – Hardware platforms used in this chapter.

| Name | Model | #Cores | Freq | L1 | L2 | LLC |
|---|---|---|---|---|---|---|
| *Haswell* | E5-2680 v3 | 12 | 2.5GHz | 32KB | 256KB | 30MB |
| *Ivy* | E5-2680 v2 | 10 | 2.8GHz | 32KB | 256KB | 25MB |

We evaluate GLS and GLK on various microbenchmarks and software systems on two modern multi-core platforms from Intel. We detail the characteristics of these platforms in Table 5.1. Based on microbenchmark results, we show that GLS adds low overheads compared to directly using locks. Additionally, we show that GLK is always able to capture the needs of the underlying workload, thus adapting to the best algorithm for each workload phase. We plug GLK in various systems: HamsterDB, Kyoto Cabinet, Memcached, MySQL, and SQLite, by overloading the `pthread` mutex library. We improve the performance of these systems by 23% on average, with essentially zero effort. Finally, using the debugging facilities of GLS, we detect two potential correctness issues in Memcached.

The main contributions of this chapter are as follows:

- GLS, a middleware locking service that simplifies lock-based programming;
- GLK, a practical lock algorithm that dynamically adapts to the contention of the underlying workload;
- Efficient implementations of GLS and GLK in our locking libraries – available at `https://lpd.epfl.ch/site/gls`;
- A novel approach for dynamically detecting correctness issues in lock-based systems.

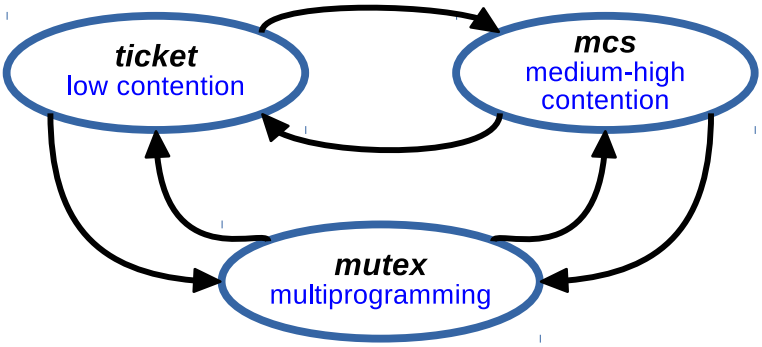



Figure 5.2 – The three modes of GLK.

Of course, neither GLS, nor GLK are silver bullets. GLS adds both latency and memory overheads compared to plain locking. Additionally, the adaptiveness of GLK adds a low performance overhead compared to directly using the corresponding lock algorithm. Therefore, a fully-customized system, for a fixed workload configuration, with every lock object set to the correct lock algorithm, will inevitably be slightly faster than with GLK. Finally, one can devise scenarios where GLK will be frequently switching modes. However, we never observe such behavior in practice.

The rest of the chapter is organized as follows. We introduce GLK in §5.2 and use it in GLS in §5.3. We use GLS and GLK to simplify and optimize modern software systems in §5.4 and conclude the chapter in §5.5.

## 5.2 GLK: A Generic Lock Algorithm

As we explained in Section 2.3, different lock algorithms are suitable for different workloads. Simple spinlocks (e.g, ticket locks) are the fastest locks under low contention. Queue-based spinlocks are more suitable under high contention. For instance, MCS locks were recently introduced in the Linux kernel to improve scalability of highly-contended locks [26]. Nevertheless, spinlocks cannot properly handle multiprogrammed environments. Due to busy waiting, threads waiting behind a spinlock might occupy a hardware context instead of a thread that could perform some useful work. This phenomenon is exacerbated in fair locks,[1] where the thread that is next in acquiring the lock might not be scheduled.

Accordingly, we design the *generic lock* algorithm (GLK), based on the premise that there is no one-size-fits-all lock algorithm. GLK adapts to the workload in order to select the most suitable way of waiting behind a busy lock. In brief, GLK collects contention information and, based on this information, periodically adapts the mode it operates in, between *ticket*, *mcs* and *mutex* (Figure 5.2). We use TICKET instead of other simple spinlocks, as TICKET is fair

[1] Fair locks, such as ticket locks, offer FIFO ordering of lock acquisitions. Queue-based locks are FIFO by design because they are implemented as queues.

and more scalable than TAS and TTAS [40]. In what follows, we first describe the design of GLK and then perform a sensitivity analysis for the configuration parameters of GLK on our target platforms.

```
typedef struct glk {
  glk_type_t lock_type;
  glk_ticket_lock_t ticket_lock;
  glk_mcs_lock_t mcs_lock;
  glk_mutex_lock_t mutex_lock;
  uint32_t num_acquired;
  uint32_t queue_total;
} glk_t;
```

Listing 5.1 – The GLK structure.

**The GLK structure.** Listing 5.1 includes the code for the GLK structure. The structure contains a `lock_type` flag that indicates the current mode of the lock, the three lock objects for *ticket*, *mcs*, and *mutex*, and two counters for gathering statistics (`num_acquired` and `queue_total`). The former counter measures the number of completed critical sections, while the latter contains the amount of queuing behind the lock.

**Measuring contention.** GLK contains a configuration parameter on how often to collect contention statistics. On spinlock-mode (i.e., *ticket* or *mcs*), we measure contention as the amount of queuing behind the lock.

Ticket locks provide this queuing information by design [111]. A ticket lock comprises two counters: `ticket` and `owner`. To acquire the lock, the thread grabs a ticket `t` by atomically incrementing (and fetching) the `ticket` field. It then spins until `t` becomes equal to `lock->owner`. To release the lock, the owner simply increments the `owner` field. Consequently, `lock->ticket` − `lock->owner` shows how many threads are waiting behind the lock (plus one for the current owner).

MCS creates a queue of waiting nodes. To measure the amount of queuing, we count the number of nodes while traversing the queue. It is important that this traversal is infrequent, because it breaks the "each node is accessed by a single thread" design goal of MCS.

Finally, the aforementioned queuing information is not sufficient for detecting multiprogramming. Multiprogramming does not relate to the contention of a single lock, but rather to the overall processor load. In other words, multiprogramming might be caused by other applications executing on the machine. Accordingly, to detect multiprogramming, GLK compares the number of running tasks to the number of available hardware contexts. On the first GLK invocation, a background thread is spawned. This background thread is shared across all GLK objects in a system and checks whether there is oversubscription of threads to hardware contexts at the system level, and whether GLK locks must switch to *mutex* mode.

**Selecting the GLK mode.** We perform a sensitivity analysis on when TICKET is faster than MCS (see §5.2.1). TICKET is consistently faster than MCS when up to three concurrent threads are accessing the lock. We thus configure the transition from *ticket* to *mcs* to happen when the amount of average queuing on a lock is more than three. To avoid frequent, unnecessary transitions, we configure the opposite transition, from *mcs* to *ticket*, to happen when queuing drops below two. Additionally, we keep the exponential moving average of the statistics in order to hide possible short-term workload fluctuations.

The periodic background thread that detects multiprogramming wakes up approximately every 100 us. If the thread detects multiprogramming, it sets a library-wide flag to inform locks that they must switch to *mutex* when they next try to adapt. However, locks switching to *mutex* mode might cause a system-load decrease, as threads block on *mutex*. Thus, the locks might continuously fluctuate between *mutex* and the other modes. To avoid this effect, we detect and avoid consecutive transitions from *mutex* to spinlocks, by exponentially increasing the number of consecutive rounds with no oversubscription required to switch away from *mutex*.

Additionally, locks that face close-to-zero contention do not cause a problem on multiprogramming. In fact, these locks should be simple spinlocks in order to complete these critical sections as fast as possible. Therefore, GLK objects that operate with minimal queuing do not switch to *mutex*, but remain in *ticket* mode.

We implement a version of MUTEX for GLK that incorporates the collection of statistics required for adaptation. We also modify the lock to support deadlock detection. Our MUTEX implementation is more lightweight than the one in the `pthread` library, as it does not include the various sanity checks of the latter. These sanity checks (and more) are provided by GLS in debug mode (see §5.3.2).

**Adapting the GLK mode.** GLK contains a configuration parameter on how often adaptation must be attempted. When adaptation is triggered, the thread currently holding the lock checks the lock statistics and decides which GLK mode to use.

To acquire the lock, a thread (i) accesses the `lock_type` flag, (ii) acquires the corresponding "low-level" lock object, (iii) checks that `lock_type` has not been modified (if it has been modified, the thread releases the low-level lock object and restarts), and (iv) performs the adaptation (if needed). This approach has the benefit that the lock and unlock functions of TICKET, MCS, and MUTEX can be used almost unmodified. The only modifications in their lock functions are to update the queuing and the number of completed critical sections statistics.

The GLK lock function is presented in Listing 5.2. The `glk_try_adap` function first checks whether it should try to adapt the GLK's mode and then checks the statistics to decide on which mode the lock will execute on. If a mode-transition happens, `glk_try_adap` returns true, hence the lock operation is restarted (line 15).

```
1  void glk_lock(glk_t* lock) {
2    do {
3      glk_type_t curr_type = lock->lock_type;
4      switch(curr_type) {
5      case TICKET_LOCK:
6        glk_ticket_lock(lock);
7        break;
8      case MCS_LOCK:
9        glk_mcs_lock(lock);
10       break;
11     case MUTEX_LOCK:
12       glk_mutex_lock(lock);
13     }
14
15     if (lock->lock_type == curr_type && !glk_try_adap(lock, curr_type))
16       break;
17     else glk_unlock_type(lock, curr_type);
18   } while (1);
19 }
```

Listing 5.2 – The GLK lock function.

**Correctness.** The correctness of GLK is not obvious at a first glance. To understand GLK, we need to consider all possible interleavings of executions. First, we can differentiate between executions with and without concurrent adaptation(s). In the latter case, without any GLK adaptation, GLK is trivially correct because it acts exactly as the low-level lock indicated by `lock->lock_type`.

Then, we will show that only a single thread $t$ can succeed the if statement in line 15 of Listing 5.2 at a time. If $t$ succeeds `lock->lock_type == curr_type`, then no other thread can succeed this statement before $t$ either adapts the lock, or releases the low-level lock. Any other concurrent thread that uses the same lock type as $t$ will block at the low-level lock. All other concurrent threads will fail the statement. Accordingly, just $t$ (i.e., one thread at a time) has the chance to trigger adaptation and enter the critical section. If no adaptation happens, `glk_try_adap` returns false and $t$ enters the critical section. Otherwise, `glk_try_adap` returns true, thus $t$ releases the low-level lock and re-runs the while loop. In that case, any thread can take the position of $t$ in succeeding the first clause of line 15, without breaking the correctness of our algorithm.

**Including additional lock algorithms.** In GLK, we chose to use a minimum number of algorithms to cover the needs of most workloads. As such, we included a TICKET spinlock algorithm for low contention cases, an MCS queue lock algorithm for contended locks and a MUTEX lock algorithm in order to cope with oversubscription of threads to hardware contexts. In our experience, and as we show in our evaluation of GLK, these locks are sufficient for a wide range of workloads. However, additional lock algorithms can be included in GLK: By modifying the lock selection algorithm previously described and by introducing new selection criteria, users can add more specialized lock algorithms to address cases where such algorithms could yield better performance (e.g., cohort locks [46]).

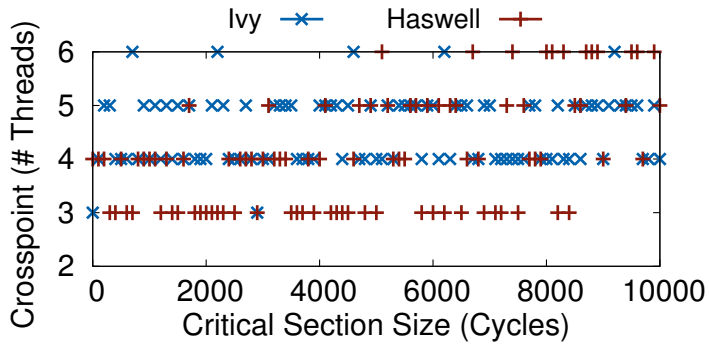



Figure 5.3 – Performance crosspoint: The number of threads that should be concurrently accessing a lock object so that MCS outperforms TICKET.

### 5.2.1 GLK sensitivity analysis

We perform a sensitivity analysis of the three most important parameters of GLK: (i) the contention thresholds that control the switch from/to *ticket* mode, (ii) adaptation period, and (iii) queue sampling period. Based on our analysis, we set the default GLK settings, which we believe are suitable for most workloads and x86 hardware platforms (in our evaluation we did not need to change any of the settings). However, our scripts for this sensitivity analysis can be used to fine-tune GLK for specific hardware platforms and workloads, if necessary.

**Ticket vs. mcs mode.** Figure 5.3 shows how many threads must contend for a single lock in order for an MCS lock to outperform a TICKET lock. On both of our platforms, TICKET is typically faster than MCS on up to three threads. Accordingly, we configure GLK in our experiments to switch from *ticket* to *mcs* mode when the average queuing behind the lock is more than three.

**Adaptation and queue sampling periods.** Figure 5.4 shows the relative throughput of GLK compared to GLK with adaptation disabled, in the extreme scenario of empty critical sections.

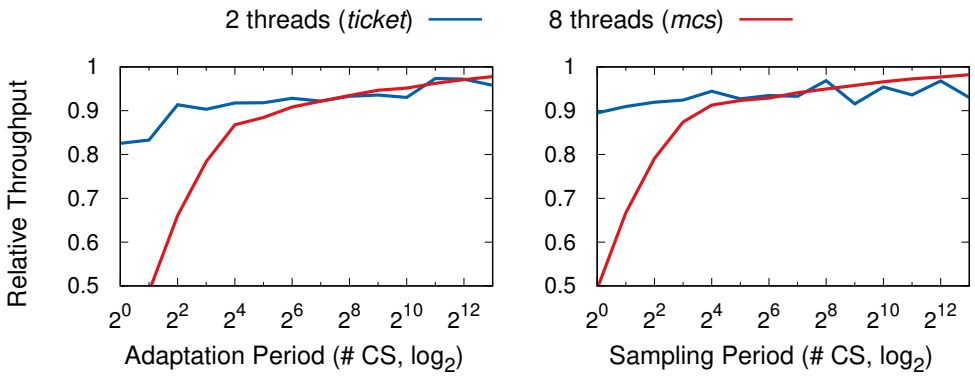



Figure 5.4 – Relative throughput of GLK in *ticket* and *mcs* modes compared to GLK with adaptation disabled, depending on the adaptation period (left) and the queue sampling period (right).

For the 2-thread execution we fix the non-adaptive GLK to *ticket* mode, while for the 8-thread execution to *mcs* mode. For both experiments the results show that – as expected – short adaptation periods incur a high performance overhead. In both cases, the results stabilize as we increase the adaptation period. Accordingly, in our experiments, we set the adaptation period to 4096 critical sections and the sampling period to 128 critical sections. With these settings, we obtain a sufficient number (4096/128 = 32) of queuing samples per adaptation.

### 5.2.2 GLK evaluation

We start by evaluating the overhead of GLK due to its adaptiveness. We then compare the performance of GLK with TICKET, MCS, and MUTEX on a set of microbenchmarks. In §5.4, we plug in and compare the performance of these algorithms in software systems. Before we proceed with the evaluation, we describe the experimental settings in our microbenchmarks.

**Experimental methodology.** Threads execute in a loop, performing lock and unlock operations on lock object(s). On every run, we configure (i) the number of threads, (ii) the number of lock objects, and (iii) the duration of the critical section (in CPU cycles). Furthermore, after every loop iteration, threads wait for a short duration to avoid long runs [112]. On every loop iteration, each thread selects a lock object at random. Our results use the median value of 11 repetitions of 10 seconds each. We do not pin threads to cores, but let the OS do the scheduling. Additionally, for fairness and for avoiding false cache-line sharing, we pad all locks to 64 bytes (one cache line). Due to space limitations, we only show the results on our *Haswell* machine (see Table 5.1). We get very similar results on our *Ivy* machine.

**Overhead.** We evaluate the overhead of GLK due to its adaptiveness. Compared to the vanilla TICKET, MCS, and MUTEX algorithms, GLK additionally executes the following steps: (i) it accesses the `lock_type` flag, (ii) it increments the `num_acquired` counter on almost every critical section,[2] (iii) it updates the queuing statistics of the lock every 128 critical sections, (iv) it tries to adapt the lock type every 4096 critical sections, and (v) it checks the `lock_type` flag for a second time. Additionally, the conditional statement on whether adaptation should be attempted is executed on every lock acquisition.

Figure 5.5 shows the throughput of GLK on three distinct configurations, each suitable for TICKET, MCS, and MUTEX, respectively. On the single-thread execution, all acquires and releases are local and uncontested (we use empty critical sections). GLK operates on *ticket* mode but is 22% slower than TICKET, mainly because of the switch statement on the `lock_type` when locking and unlocking. Even if we turn adaptation off (i.e., overhead steps (ii)-(iv)), GLK is still 19% slower than TICKET. This comparison reveals an inherent overhead of GLK: An adaptive lock must access a flag to find the lock type to use. Still, GLK delivers 24% and 30% more throughput than MCS and MUTEX, respectively.

[2] In TICKET mode, we take advantage of the `current` counter of the lock to avoid incrementing the `num_acquired` counter.

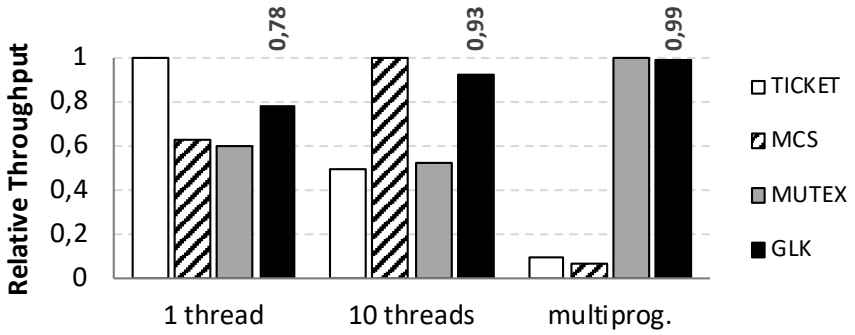



Figure 5.5 – Relative throughput of GLK compared to the best per-configuration lock on various configurations.

The second workload includes 10 threads and empty critical sections, a configuration suitable for MCS locks. In this case, GLK is 7% slower than MCS. Again, even if we remove adaptation from GLK, GLK is still 6% slower than MCS. The overhead in *mcs* mode is lower than in *ticket*, because there is actual contention behind the lock, thus the overhead is partly hidden by waiting. Again, regardless of the overhead of GLK, GLK is still significantly faster than TICKET and MUTEX, respectively.

Finally, the third configuration of Figure 5.5 involves 10 threads and multiprogramming (we initialize 48 additional threads that just spin locally). In this configuration, GLK is only 1% slower than MUTEX, but much faster than TICKET and MCS. As we have pointed out, fair locks suffer on multiprogrammed workloads.[3]

**Single lock behavior.** We evaluate the behavior of a single lock as the number of concurrent threads increases. Critical section are 1024 cycles long. Figure 5.6 includes the results of

[3] There do exist techniques, such as time-published queue-based locks [68], for alleviating this problem.

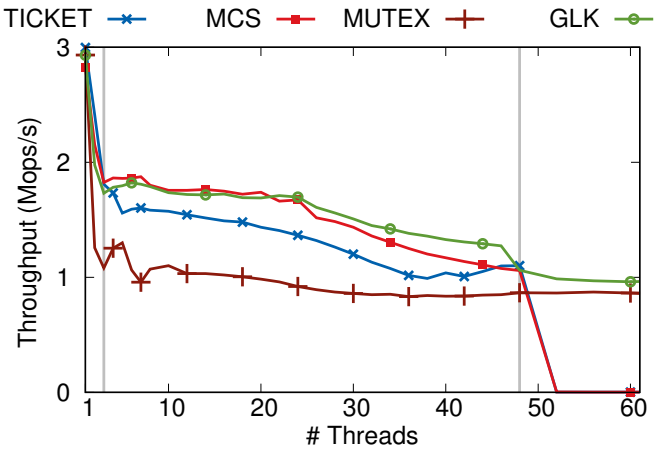



Figure 5.6 – A single lock on varying contention.

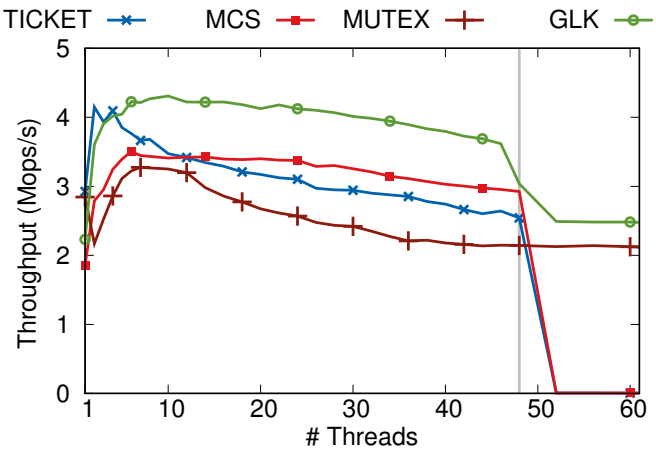



Figure 5.7 – Eight locks on varying contention.

this experiment. As expected, on a low thread count (i.e., up to 3 threads), TICKET is the fastest lock. On these executions, GLK operates in *ticket* mode and follows the performance of TICKET closely. As we increase the contention further, GLK switches to *mcs* mode, thus behaving similarly to MCS. Finally, when using more than 48 threads, where there is oversubscription of threads to hardware contexts, GLK executes in *mutex* mode, in order to handle multiprogramming.

**Multiple locks behavior.** We experiment with eight locks as the number of concurrent threads increases, using critical sections of 1024 cycles. On every iteration, each thread selects one of the locks at random, using a zifpian skewed distribution with alpha set to 0.9. In other words, some locks are more frequently utilized than the rest. The two most busy locks serve 34% and 18% of the requests, respectively. Intuitively, in a software system, some locks might be more contended than others: This is one of the cases that we aim at capturing with GLK. The developer must not have the difficult duty of identifying contended locks and customizing their algorithm accordingly.

Figure 5.7 includes the results of this test. For up to three threads, all eight locks face low contention. Thus, TICKET and GLK (in *ticket* mode) are the fastest. For more threads, the contention on one to two locks increases, thus MCS is the most suitable choice. GLK is able to adapt to *mcs* mode only these highly-contended locks, while keeping the rest in *ticket* mode. This behavior results in GLK being approximately 20% faster than MCS on the non-multiprogrammed configurations. Under multiprogramming, GLK uses *mutex* mode for the two contended threads, while the rest remain in *ticket* mode.

**Varying workload.** We evaluate the behavior of a single lock when the contention varies over time. More precisely, the execution is broken into phases of 0.5-1s.[4] In each phase, the

[4] Note that GLK is able to adapt with much more fine-grain granularity. Assuming 1M acquires/s, adaptation happens approximately every 4 ms, hence the phase length can be as low as roughly 10 ms.

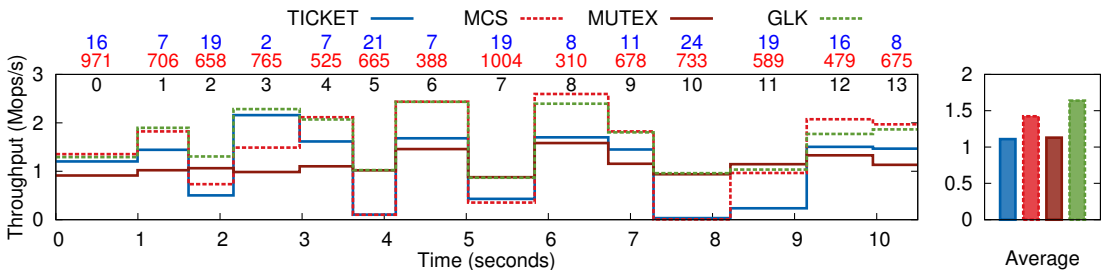



Figure 5.8 – One lock under varying contention levels over time. The numbers on top of the graph represent the number of threads on each phase (blue, top), the critical section duration in cycles (red, middle), and the phase id (black, bottom).

number of threads that execute is selected at random from 1-24. Additionally, 30 background threads run on the processor. These background threads represent other applications that could be executing on the same machine.

Figure 5.8 shows the throughput of different lock algorithms as a parameter of time and per-phase configuration. On average, GLK delivers 15% higher throughput than the second fastest lock, namely MCS. GLK achieves this by dynamically adapting its configuration during every phase, depending on the needs of the workload. For instance, in phase 3, where contention is very low, GLK switches to *ticket* mode, thus delivering the same performance as TICKET. In contrast, in phases 0-1, where contention is very high, GLK switches to *mcs* to better scale with the number of contending threads. Finally, in multiprogrammed phases, such as 2 and 10, GLK switches to *mutex* in order to avoid potential performance degradation caused by busy waiting.

**Conclusions.** GLK can successfully adapt to the needs of the underlying workload at runtime, in order to deliver performance that is close to the best lock algorithm at any point in time. Of course, due to the adaptation overhead, GLK is usually slightly slower than the best per-configuration lock algorithm for fixed workloads. Based on these results, we claim that GLK delivers close-to-optimal performance for any workload and configuration combination. Additionally, as we show in §5.4, in systems with many locks and complex interactions, GLK can outperform any lock algorithm precisely because of its adaptiveness.

## 5.3 GLS: A Generic Locking Service

GLS is a *generic locking service* that simplifies lock-based programming by handling many of the complexities that developers must typically cope with when using locks. GLS provides the classic lock interface (e.g., functions for locking and unlocking). However, unlike traditional lock libraries, GLS accepts any arbitrary memory address (or even value) as an input parameter to `gls_lock`. With GLS, `gls_lock(17)` is a valid lock invocation. GLS maps the input address

| Function | Description |
|---|---|
| `gls_init()` | Initialize GLS |
| `gls_destroy()` | Stop GLS and cleanup |
| `gls_lock(void* m)`<br>`gls_trylock(void* m)`<br>`gls_unlock(void* m)` | Lock, trylock, or unlock `m` using GLK algorithm |
| `gls_A_lock(void* m)`<br>`gls_A_trylock(void* m)`<br>`gls_A_unlock(void* m)` | Lock, trylock, or unlock `m` using algorithm `A`. `A` can be `tas`, `ttas`, `ticket`, `mcs`, `clh`, or `mutex` |
| `gls_free(void* m)` | Remove `m` from GLS |

Table 5.2 – GLS interface.

to a lock object behind the scenes. Accordingly, the user of GLS does not need to worry about declaring and initializing locks.

Additionally, GLS offers two very useful extensions: (i) a debugging extension that can detect the most common bugs in lock-based programming, and (ii) a profiler that reports the amount of contention on each lock. We first describe the default design and implementation of GLS and then we detail the debugging and profiler extensions of GLS.

### 5.3.1 Programming with GLS

**Interface.** Table 5.2 presents the interface of GLS. Apart from the initialization functions, GLS includes various calls for locking and unlocking using different algorithms. These functions accept any arbitrary value as an input, except for `NULL`. The default interface of GLS (`gls_lock`) utilizes the GLK algorithm. In addition, GLS offers an explicit interface to six other algorithms.

**Implementation.** GLS is essentially a cache for locating the lock object that corresponds to an address. We implement GLS on top of a modified version of the lock-based CLHT hash table [41]. CLHT has several properties that are necessary in GLS: (i) it uses cache-line-sized buckets, hence operations typically complete with at most one cache-line transfer, (ii) searching for a key is a read-only, wait-free operation, (iii) failing to insert a key is also read-only and wait-free, and (iv) it is resizable. Consequently, when the lock objects used in a system are stable (i.e., there are not many allocations and deallocations of locks), the CLHT hash table in GLS becomes a read-mostly hash table, thus incurring low overhead. The workflow of `gls_lock` is as follows:

```
1  int gls_lock(void* addr) {
2    glk_t* lock = (glk_t*) gls_clht_put(addr);
3    return glk_lock(lock);
4  }
```

In line 2, GLS is searching in the hash table for the lock object that corresponds to the given address. We modify `clht_put` to create and initialize a new lock object for `addr` if `addr` is

not found. If `addr` already exists in the hash table, then the corresponding lock object is returned. The `gls_unlock` function uses `gls_clht_get` to fetch the lock which maps to the given address. (As we show in §5.3.2, if `gls_clht_get` returns `NULL`, GLS detects that an uninitialized lock is used in unlock.) The `gls_A_lock` functions perform the same workflow as `gls_lock`, but initialize and use the lock function of the corresponding algorithm `A`.

**Lock-cache optimization.** In locking, the most common pattern involves acquiring and later releasing the same lock, without accessing any other locks in the meantime (the opposite case is called *lock nesting*). Additionally, there is temporal locality of accesses: A lock that is used by a thread will be reused in the near future by the same thread with high probability. To optimize for these patterns, we introduce a single object cache in GLS. This cache keeps track of the address and the lock object of the latest lock that has been accessed. If a lock/unlock operation finds the target address in the cache, there is no need to access the GLS hash table. On a cache miss during locking, the cache is updated with the target address-lock object pair.

**Memory overhead.** GLS adds memory overhead over traditional locking, mainly due to the hash table. CLHT keeps up to three key-value pairs per cache line (64 bytes), hence the minimum overhead introduced by GLS is one-third of a cache line per lock. In our experience, the CLHT used in GLS typically achieves 60-70% occupancy, thus we estimate the overhead per lock to be approximately 32 bytes, or 50% of a cache-line-sized lock object.

**Performance overhead.** We evaluate the performance overhead of GLS compared to using locks directly.

*Single Thread.* Figure 5.9 depicts the latency overhead of GLS on a single thread, depending on how many locks are accessed. On each iteration, the thread picks a lock at random. With a single lock, the overhead of GLS is just a few cycles due to the lock cache that is always able to locate the lock without accessing the GLS hash table. The same applies for the unlock latencies in this experiment, regardless of the number of locks: Unlock operations always hit on the lock

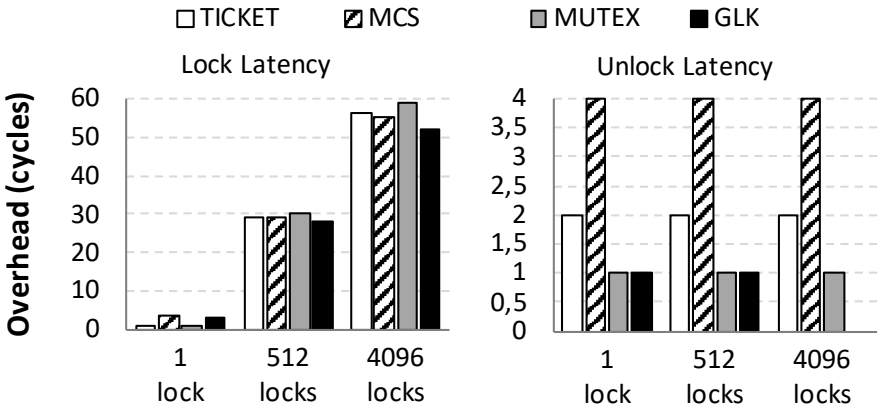



Figure 5.9 – Latency overhead of GLS over directly using locks on a single thread.

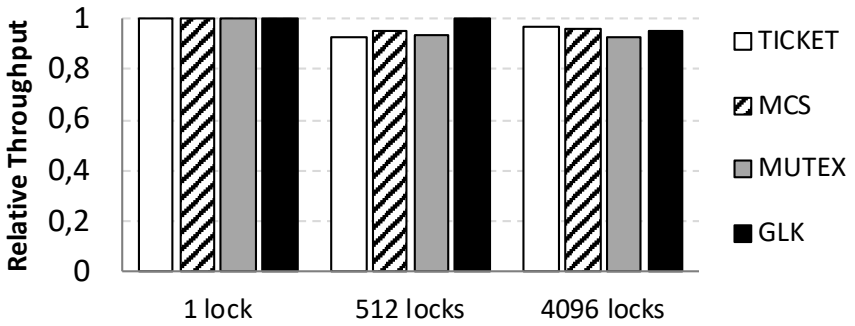



Figure 5.10 – Relative throughput of GLS over directly using locks on 10 threads.

cache (no lock nesting). Without the cache, the unlock latency overhead is close to the GLS overhead in lock functions. With 512 locks, GLS adds approximately 30 cycles overhead, which corresponds to roughly 100% increase in lock latency. As we increase the number of locks, the size of the hash table does not fit in the L1 cache, thus the overhead of GLS increases.

*Multiple Threads.* Figure 5.10 compares the throughput of different lock algorithms when used in GLS to direcly using the lock algorithm, with 10 threads competing for 1, 512, or 1024 locks (high, medium and low contention respectively). Each thread randomly chooses among the locks and spends 1024 cycles in the critical section. When locks are not contested (4096 locks to choose from), the overhead of GLS is proportional to the duration of the critical section. In the presence of contention, however, the overhead of GLS can be masked by waiting.

### 5.3.2 Debugging with GLS

GLS can be configured to detect several lock-related issues. In order to detect potential bugs, including deadlocks, GLS in debug mode keeps track of the owner of each lock object. In what follows, we describe the issues that GLS can detect. Note that some of these issues could be seen as features depending on the semantics of the specific lock algorithm in use.

**Uninitialized locks.** GLS handles the mapping of addresses to lock objects. Accordingly, GLS can detect when a thread is trying to access an uninitialized lock. Upon releasing a lock, uninitialized locks are detected when the target address does not exist in the hash table, which means that the lock was not acquired before. In order to detect uninitialized locks while trying to acquire a lock, GLS adds special values in the overloaded MUTEX locks for static initialization (instead of the default ones used by the `pthread` mutex). When trying to acquire a lock, if the call to `gls_clht_put` does not find an address-lock mapping, there are two possible cases: either the lock has been statically initialized, or the lock has not been initialized at all. To discern between the two, GLS checks whether the MUTEX object contains the special values used for static initialization. If that is the case, the lock was statically initialized, otherwise an error is detected. Due to the static declaration of these locks, it is certain that if not initialized, they will contain a zero value [2].

**Double locking.** GLS can check whether the current owner of a lock tries to acquire that same lock again. GLS implements this functionality by comparing the id of the thread that is performing the operation with the id of the lock owner. Of course, double locking is a subset of deadlock detection.

**Releasing a free lock.** Releasing an already free lock can either break some lock algorithms (e.g., TICKET), or result in race conditions where a thread is trying to acquire the lock while another is falsely releasing it at the same time. GLS checks whether an unlock function call operates on a lock that is already free.

**Releasing a lock with wrong owner.** GLS checks that the owner id of the lock to be released is the same as the id of the thread performing the unlock operation.

**Deadlocks.** Deadlocks can be very hard to detect and debug [55]. GLS implements a technique for detecting and reporting deadlocks at runtime. GLS implements deadlock detection in the following steps:

1. GLS augments the hash table with a *waiting* array that indicates which lock each thread is waiting on (if any).
2. Before calling the lock/trylock function in the `gls_clht_put` invocation, GLS records that the corresponding thread is waiting on the target address.
3. When the lock is acquired (or the trylock function completes), GLS updates the waiting structure to indicate that this thread is not waiting behind that lock anymore.
4. Additionally, the owner of that lock is set to the thread id. For trylock, the owner id is set iff the lock was successfully acquired.
5. On lock release, the owner id for that lock is cleared.

Based on the aforementioned steps, if the thread is waiting behind a lock for a long time (i.e., more than a second), GLS triggers the following deadlock-detection procedure:

```
// the invoking thread is waiting on wait_lock
wait_on  = wait_lock;
do {
  /* find the owner of the lock that the
     previous thread is waiting on */
  owner_id = gls_debug_get_onwer(wait_on);
  // if the invoking thread re-appears
  if (owner_id == gls_get_id())
    gls_debug_deadlock_print();
  // find the lock that owner_id is wating on
  wait_on = gls_debug_get_wait_on(owner_id);
} while (wait_on_lock != NULL);
```

In short, the invoking thread $owner_0$ tries to find a cycle with $owner_0 \rightarrow waiting_0 \rightarrow owner_1 \rightarrow waiting_1 \rightarrow ... \rightarrow owner_0$ relationships. If such a cycle exists (a series of relationships that

starts and ends with the same id) a deadlock is detected. In that case, GLS prints the details of the cycle as well as the backtrace of the call that caused the deadlock:

```
[GLS]WARNING> DEADLOCK 0x1ad0010 - cycle detected
[2   waits for 0x1ad0010] ->
[9   waits for 0x1acfff4] ->
[8   waits for 0x1acfff8] ->
[2   waits for 0x1ad0010]
[BACKTRACE] Execution path:
[BACKTRACE]#0 ./stress_error_gls(glk_lock+0x4b)
gls/src/glk.c:392
[BACKTRACE]#1 ./stress_error_gls(gls_lock+0x54)
gls/src/gls.c:196
[BACKTRACE]#2 ./stress_error_gls(test+0x248)
gls/bmarks/stress_error_gls.c:160
```

Note that this output is a simplified version of the actual output. GLS can provide more details for debugging the deadlock easier (e.g., automatically setting GDB breakpoints and printing the actual `pthread` ids of threads).

**Removing GLS deadlock-detection overhead.** The overhead of GLS's detection technique is high, because every lock and unlock operation has to update some metadata in the GLS hash table. In our microbenchmarks, GLS in debugging mode performs up to 4 times slower than without debugging. However, this metadata (regarding waiting-for relations and lock ownership) is only checked when the deadlock-detection procedure is triggered. Accordingly, we can avoid updating this metadata in normal operation and only have the threads update it when they are stuck behind a lock for a significant amount of time. With this approach, deadlock detection happens after a couple of invocations to the detection procedure, but we almost completely remove the overheads while threads operate normally.

### 5.3.3 Profiling with GLS

GLS can be configured to operate in a low-overhead profiler mode. In this mode, GLS reports per-lock statistics regarding the average queuing behind the lock, the lock acquisition latency, and the critical-section duration. For instance, in SQLite the output is similar to the following:[5]

```
[GLS] queue: 0.03 | l-lat: 96    | cs-lat: 194
@ (0x7fe6318eb660:sqlite3.c:pthreadMutexEnter)
[GLS] queue: 4.50 | l-lat: 13963 | cs-lat: 2848
@ (0x7fe6318eb4e0:sqlite3.c:pthreadMutexEnter)
```

[5] SQLite wraps mutex calls in a function. We could get the actual location of this invocation by printing the backtrace of the call.

We use this profiler mode to easily detect highly-contended locks that are likely to become a scalability bottleneck as we scale a system. For example, in §5.4, we use the profiler to better redesign locking in Memcached and to better understand the performance results on various systems.

Additionally, GLK can be configured to print the mode transitions that it performs, as well as the reason behind each transition. This output can be used to better understand potential variations in the workload behavior. It can also be used to decide on a pre-determined lock algorithm that is the most suitable for a given lock object in a system (in case the designer selects to use a per-lock custom algorithm).

## 5.4 GLS / GLK in Lock-based Systems

We modify locking in various concurrent systems in order to evaluate the effectiveness of GLS and GLK. In most systems, modifying locks is as simple as overloading the `pthread` mutex functions with our own lock implementations. We first show how we can easily employ GLS in debugging and optimizing Memcached. We then plug GLK in various modern software systems in order to improve their performance.

### 5.4.1 Re-engineering memcached with GLS

**Debugging memcached.** We notice that when we overload `pthread` mutexes with certain lock algorithms (i.e., TICKET, MCS and GLK), Memcached (v. 1.4.20) hangs. We identify this behavior as the perfect opportunity to show the debugging capabilities of GLS in action. Indeed, GLS reports the following output:

```
[GLS]WARNING> LOCK 0x6344e0 - Uninitialized lock
[BACKTRACE] #0 memcached/thread.c:662
[BACKTRACE] #1 memcached/assoc.c:72
[GLS]WARNING> UNLOCK 0x62a494 - Already free
[BACKTRACE] #0 memcached/slabs.c:836
[BACKTRACE] #1 memcached/assoc.c:249
```

The first warning is about locking the uninitialized `stats_lock` in `assoc.c`. The second one involves unlocking the `slabs_rebalance_lock` before it is ever acquired. Based on the output of GLS, we easily fix these two issues. Notice that these issues do not manifest with MUTEX, because (i) the `stats_lock` is always initialized to zero due to its static declaration [2], and (ii) unlocking a free lock with MUTEX leaves the lock in the same state. However, the first issue (locking the uninitialized `stats_lock`) is indeed a programming error, as the behavior of MUTEX in such cases is undefined [3]. The second issue (unlocking the `slabs_rebalance_lock` without acquiring it) can either be an error, or not, depending on the configuration of `pthread` mutex.

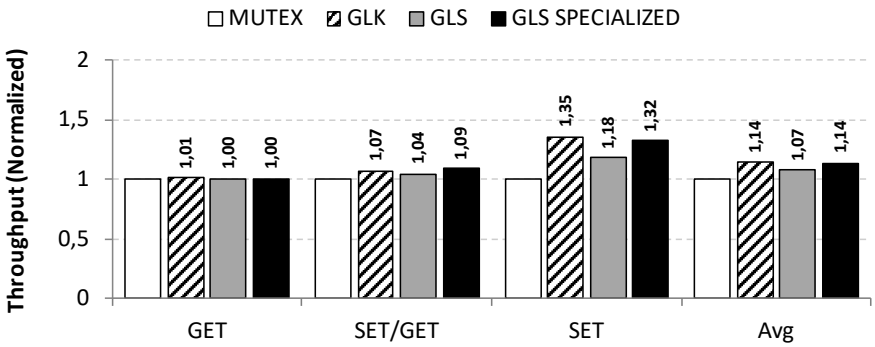



Figure 5.11 – Normalized (to MUTEX) throughput of our `memcached` implementations on *Ivy*.

**Optimizing memcached.** The main goal of GLS is to make programming with locks easier. To showcase using GLS in practice, we re-implement synchronization in the `memcached` key-value store from scratch, using the GLS API directly. We remove the code for lock declaration and initialization and replace the calls to `pthread` locks with calls to GLS. We use the `gls_lock` and `gls_unlock` functions and let GLK choose the most suitable locking technique for our workloads. We remove 26% of the synchronization code of the application and modify 186 lines of code in total. This implementation is the way we expect that most GLS users will be using the service, allowing for fast and easy development of lock-based applications.

Figure 5.11 compares the performance of Memcached when using MUTEX, GLK, as well as GLS using GLK on our *Ivy* platform. The GLS version is 7% slower than directly using GLK. This is due to the overheads of GLS. Still, the GLS version is 7% faster than the default Memcached implementation with MUTEX.

We then set out to better understand the performance of locking in `memcached`. We first observe the output of GLK and notice that most locks operate in TICKET mode, which hints at these locks facing little contention. We then use the GLS profiler mode (see §5.3.3) to understand the different requirements and behavior of each lock. What we discover is that all the locks used in `memcached` exhibit low contention, with the exception of specific global locks (e.g. the `stats_lock`).

Accordingly, we devise a second version of `memcached`, again using GLS. This time we use the explicit interface of GLS (see Table 5.2), choosing the lock algorithm that best suits each lock. We use MCS locks for the contended global locks and TICKET locks for the internal hash table and all other locks. GLS allows us to use any lock algorithm by simply modifying the lock and unlock function calls. This enables us to avoid the adaptation overheads and tailor our synchronization code to the optimal strategy. Figure 5.11 shows that the performance gains from this implementation are significant. Specifically, GLS SPECIALIZED achieves 14% higher throughput than the default MUTEX version, the same as GLK. Programmers with experience in lock-based programming can use this interface to achieve higher performance while still benefiting from the simplicity of GLS.

| | |
|---|---|
| **HamsterDB [140]** | |
| Version: 2.1.7<br># Threads: 2 | An embedded key-value store. We run three tests with random reads/writes, varying the read-to-write ratio among 10% (WT), 50% (WT/RD), and 90% (RD). |
| **Kyoto [97]** | |
| Version: 1.2.76<br># Threads: 4 | An embedded NoSQL store. We stress Kyoto with a mix of operations for three database versions (CACHE, HT DB, B+-TREE). |
| **Memcached [62]** | |
| Version: 1.4.22<br># Threads: 8 | An in-memory cache. We evaluate Memcached using a Twitter-like workload [103]. We vary the get ratio from 10% (SET), 50% (SET/GET), to 90% (GET). We dedicate one socket on each machine to the Memcached server and the other to the clients. |
| **MySQL [125]** | |
| Version: 5.6.27 | A relational DBMS. We use Facebook's LinkBench and tune the MySQL server following Facebook's guidelines [56] for in-memory (MEM) and SSD-drive (SSD) configurations. |
| **SQLite [154]** | |
| Version: 3.8.5 | A relational DB engine. We use TPC-C with 100 warehouses varying the number of concurrent connections (i.e., 8, 16, 32, and 64). |

Table 5.3 – Software systems and configurations.

### 5.4.2 Optimizing systems with GLK

We modify locking in five software systems. We select the set of systems so that they employ locking in diverse ways, including concepts such as global locking, fine-grained locking, reader-writer locks, and conditional variables. Table 5.3 includes a short description of the systems that we use, as well as the workloads that we use to evaluate them. All of our experiments use a dataset size of 10 GB, except for the MySQL SSD configuration. For this experiment, we use a dataset of 100 GB. We tune each system to achieve maximum throughput and configure the number of threads used based on the maximum-throughput configuration with their default MUTEX locks. Figures 5.12 and 5.13 show the throughput of the target systems when employing different lock algorithms on our *Ivy* and *Haswell* platforms respectively.

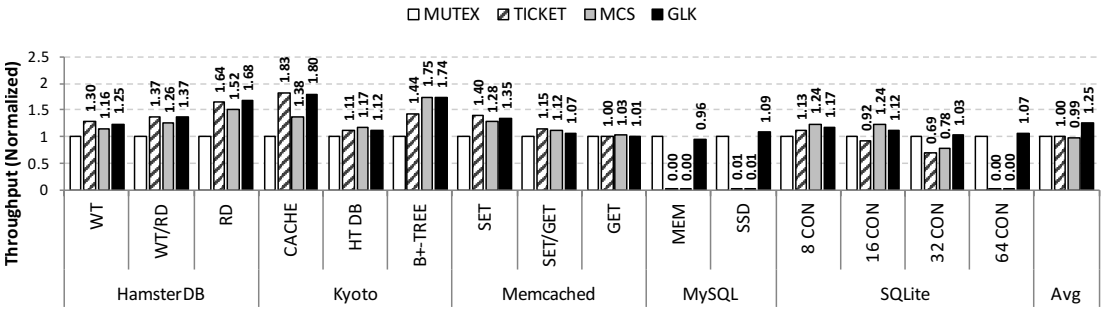



Figure 5.12 – Normalized (to MUTEX) throughput of various systems with different locks on our Ivy machine.

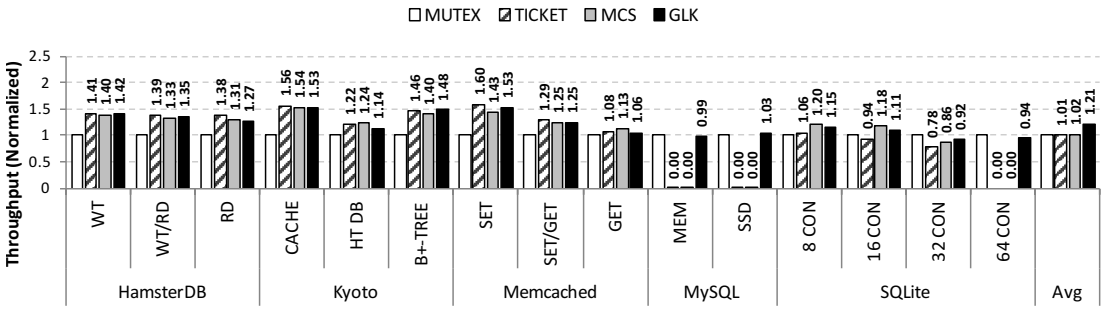



Figure 5.13 – Normalized (to MUTEX) throughput of various systems with different locks on our Haswell machine.

**Overall.** For both platforms, we see that in 14 out of 15 configurations for *Ivy* (and 12 out of 15 for *Haswell*), GLK improves the performance over the default MUTEX lock on the target systems. The performance gains range from 1% to 80% on our *Ivy* machine, and from 3% to 53% on our *Haswell* machine. On average, GLK delivers 25% higher throughput than MUTEX on *Ivy* and 21% higher on *Haswell* by selecting the most appropriate locking technique per lock, per phase, and per configuration. Naturally, there do exist configurations where spinlocks are sufficient. In these cases, the performance of both TICKET and MCS is similar. Additionally, there are few configurations where MUTEX delivers the highest throughput. In these configurations, where GLK does not deliver the highest throughput, GLK is slower only up to 8% than the best performing lock. In contrast, in the configurations where GLK delivers the best performance, we see the power of adaptiveness, as no static algorithm can capture the characteristics of the workload. The overall trends in the results are intuitive. In low-contention configurations, TICKET (i.e., simple spinlocks) delivers the best performance due to its simplicity. On higher contention levels, MCS is the fastest. Finally, in multiprogrammed configurations, blocking locks, such as MUTEX, are necessary.

**HamsterDB.** The HamsterDB embedded key-value store [140] relies on a global lock. Of course, the contention on that lock is very high. We measure with the GLS profiler that with $N$ worker threads, the average queuing behind the lock is always close to $N-1$. Consequently, we use just two threads as the application cannot scale further. TICKET delivers the best performance. GLK operates in *ticket* mode, delivering throughput very close to TICKET. MUTEX lags behind in performance because it employs, unnecessary for this workload, block and unblock invocations.

**Kyoto Cabinet.** The Kyoto Cabinet NoSQL store [97] comes in two flavors: a hash table, and a B+-tree-based implementation. The hash-table version has two extensions, a store (HT DB) and a cache (CACHE). All three versions protect the main data structure with a highly-contended global reader-writer lock.[6]

[6] For TICKET, MCS, and GLK, we overload the `pthread` reader-writer locks with our custom TTAS-based implementation.

Additionally, the hash-table versions use 16 mutexes, each protecting a group of buckets. These locks typically face very low contention: We use GLS in profiler mode to measure the contention and discover that the average queuing behind locks is less than 0.1 and 0.05 for CACHE and HT DB, respectively. However, the throughput of CACHE is approximately 10-times higher than that of HT DB, which means that there is significantly higher traffic on the locks of the former. Additionally, CACHE utilizes up to 10 levels of lock nesting. Nesting with MCS locks is expensive because, for each lock, threads must find and use a separate queue node. These behaviors are reflected in the results of CACHE, where TICKET is significantly faster than MCS. For CACHE, GLK operates in *ticket* mode and thus delivers throughput very close to TICKET. On HT DB, locks are accessed less frequently, hence the performance gains obtained by changing the locks used are smaller than those on CACHE.

The tree-version uses reader-writer locks for the nodes of the tree and mutexes for a custom cache of the tree nodes. These mutexes are highly contended: Both MCS and GLK significantly outperform TICKET and MUTEX.

**Memcached.** As we describe earlier in this chapter (Section 5.4.1), the locks in our Memcached experiments typically face low contention. Thus, TICKET delivers the highest performance for `memcached`. Consequently, with GLK, most locks operate in *ticket* mode, achieving performance that is very close to that of TICKET for both platforms.

**MySQL.** MySQL is a complex system that handles most low-level synchronization with custom locks (semaphores). Clearly, neither simple, nor queue-based spinlocks are sufficient for MySQL: In both workloads, MySQL oversubscribes threads to hardware contexts. The result is a livelock for both MCS and TICKET that deliver less than 100 operations per second. Notice that the fairness of these two locks exacerbates the problem. In comparison, a TTAS lock – not shown in the graph – gives 90% and 50% lower throughput than MUTEX on the MEM and SSD workloads, respectively.

On MySQL, we see an inherent limitation, but also the true power of adaptiveness. On the in-memory workload, GLK is 4% slower than MUTEX on *Ivy* and 1% slower on *Haswell*. This difference in throughput is due to the adaptiveness overhead. Directly using the mutex implementation of GLK results in the exact same throughput as MUTEX. However, on the SSD workload we see the power of adaptiveness: Many locks in MySQL are lightly contended, thus using *ticket* mode instead of *mutex* results in 9% higher throughput on *Ivy* and 3% higher throughput on *Haswell*. In complex systems, such as MySQL, it is nearly impossible to manually customize every single lock with the "correct" algorithm.

**SQLite.** SQLite is based on a B-tree data structure. SQLite implements concurrency with both coarse and fine-grained locks. SQLite uses a MUTEX for each database (e.g., each new connection), another for memory allocation, and a last one for protecting the database cache. However, the nodes of the B-tree are protected by custom reader-writer locks. The mutexes of SQLite become contended as we increase the number of connections.

With 8 and 16 connections, MCS gives the best performance, with GLK following closely. However, on 32 connections, the workload has some phases with multiprogramming, thus the performance of MCS and TICKET drops. As expected, on 64 connections, using fair spinlocks results in livelocks. In comparison, TTAS – not shown in the graph – delivers 30% lower throughput than MUTEX. On both 32 and 64 connections, GLK achieves slightly better throughput than MUTEX on *Ivy* and follows closely MUTEX on *Haswell*, with lightly contended locks remaining in *ticket* mode.

**Conclusions.** GLK is able to adapt and capture the needs of all the 15 workloads on the five systems that we evaluate. Doing so, GLK improves the performance of these systems by 25% and 21% on average on our two platforms, with practically zero effort on the developer's side. Even on the configurations that GLK does not deliver the best performance, due to the overheads of adaptation, GLK is only up to 8% slower than the best performing lock. Consequently, our experimental results validate our claim that GLK can deliver close-to-optimal performance regardless of the configuration.

## 5.5 Conclusions

In this chapter we introduced GLS, a *generic locking service* and the accompanying GLK lock algorithm. GLS is a middleware that makes lock-based system development significantly easier (i) by removing the need for manually handling lock declaration and initialization, (ii) by detecting several common lock-related correctness issues, and (iii) by profiling and reporting per-lock statistics, such as the contention behind a lock and its acquisition latency. GLK simplifies things further, by monitoring the contention levels of a lock in order to dynamically adapt the locking algorithm to the needs of the underlying workload, delivering the best performance among spinlocks, queue-based locks, and blocking locks.

We showed that GLS can simplify concurrent programming, adding low overheads compared to directly using a locking algorithm. We also showed that GLK is always able to capture the needs of the workload, adapting to the best-performing algorithm for each workload phase. We used GLS to re-implement synchronization in Memcached, resulting in 26% less lock-related code and achieving 14% higher throughput. We also used the debugging facilities of GLS to detect two locking issues in the initial Memcached implementation. Finally, we used GLK in five software systems: HamsterDB, Kyoto Cabinet, Memcached, MySQL, and SQLite, comparing against their default locking algorithm, as well as the different modes that GLK can operate in. We improved the performance of these five systems by 23% on average, on two different platforms, using GLK, with essentially zero effort.

# 6 Tuning Distributed Transactions in the Datacenter

## 6.1 Introduction

Modern hardware, with large amounts of main memory and high-speed networks with Remote Direct Memory Access (RDMA) capabilities, has already made its way into datacenters, enabling a new generation of distributed algorithms and systems. Several transactional systems have been devised to take advantage of this modern hardware [8, 32, 52, 53, 88, 106, 174]. These systems achieve high throughput, low latency and good scalability in On-Line Transaction Processing (OLTP) workloads. They can commit tens of millions of transactions per second, with latencies in the micro-second range.

However, achieving such performance is not straightforward. Implementing even a single *SELECT* statement of an OLTP workload involves at least two physical design decisions that arise from the use of RDMA networks: (1) tuning the parameters of data indexes and storage, and (2) data partitioning and remote data access optimization. Previous work has required manual tuning of such parameters to achieve performance. However, this process is both cumbersome and error-prone when performed for every parameter of a system. Additionally, even small changes in the workload, or the underlying hardware, require fully repeating the process. The impact of tuning can be significant. Figure 6.1 depicts 2.2x difference in performance between finely tuned and default physical designs when running TATP [120].

In this chapter, we present SPADE (Scale-out PhysicAl Design tunEr), a tuner for OLTP workloads on modern RDMA clusters, which automates these physical design configuration choices. First, SPADE tunes the configuration of the indexes and storage parameters of a workload (1). Maximizing index performance on modern RDMA clusters requires specific configuration based on the size of rows and the operation mix. For example, inlining data in the primary index is beneficial for small row lookups, as it reduces the number of remote accesses. In contrast, for large rows, it is better to store the data outside of the index, as this reduces the amount of data transferred [52]. The exact size threshold depends on the workload and the underlying hardware. Additional operations on the table (e.g., insertions or updates) and indexes (e.g. secondary) further complicate these decisions. The impact on performance

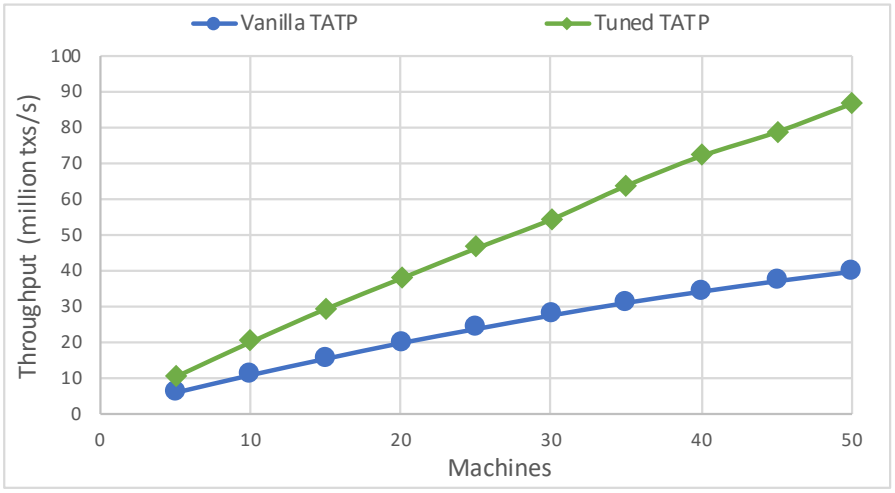



Figure 6.1 – Effect of physical design tuning on TATP.

when using an incorrect configuration is non-negligible: we noticed up to 15% impact on throughput and 22% on latency, when configuring indexes.

Additionally, SPADE optimizes partitioning and remote data accesses (2). While traditional systems rely heavily on partitioning, modern distributed transactional systems can take advantage of one-sided RDMA reads to avoid expensive network packet processing, improving throughput and lowering latency. Using RDMA reads is beneficial for a range of operations. However, message-passing (or Remote Procedure Calls – RPCs) can outperform direct accesses when data is collocated on the same machine, as well as for large write-intensive operations, as we show in this chapter. SPADE identifies whether to use RDMA or message-passing based on the schema and workload characteristics. Correctly optimizing remote accesses not only improves the throughput, but also the median and tail latency of a workload. In our experiments, tail latency is reduced by up to 80% when correctly tuning remote accesses.

SPADE tunes OLTP workloads offline. Initially, SPADE takes query plans of the stored procedures of an OLTP workload as input. By examining these plans and combining them with information on the schema and workload characteristics, SPADE identifies partitioning opportunities, collocating rows of different tables that are often accessed together. Additionally, SPADE creates a model for each procedure of the workload, consisting of low-level operations, such as single row lookups and updates. Feeding these models with the performance characteristics of the cluster, gathered through microbenchmarks, enables SPADE to decide on a configuration for each table (i.e., data and index design), as well as for each stored procedure (i.e., using message-passing or one-sided remote memory accesses).

SPADE is designed and implemented as an additional level of tuning on top of FaRM [52, 53]. FaRM is a main memory distributed computing platform that uses one-sided RDMA memory accesses, battery-backed main memory and transaction, replication, and recovery protocols to achieve high availability, scalability and performance on modern datacenter hardware. We

evaluate the performance of OLTP workloads tuned with SPADE on top of FaRM using two popular OLTP benchmarks: TATP and TPC-C. We study the extent to which tuning decisions affect performance, showing that individually, they improve throughput by up to 50% and reduce latency by as much as 25%. Combined, these decisions improve throughput by as much as 117%, while lowering latency by up to 75%. We highlight some counter-intuitive choices that SPADE makes, which would be difficult for a developer or administrator to identify. These choices improve throughput and reduce both median and $99^{th}$ percentile latency.

In summary, the contributions of this chapter are the following:

- We identify RDMA-specific tuning opportunities that are crucial to the performance of OLTP workloads on modern RDMA clusters.
- We introduce SPADE, a physical design tuner for OLTP workloads on RDMA clusters. SPADE automates index tuning, data partitioning, and remote data access optimization, based on the schema, and the workload and hardware characteristics.
- We evaluate the effect of physical design tuning on the throughput and latency of OLTP workloads on modern RDMA clusters. This evaluation is useful in its own right, since it showcases the effects of tuning outside the context of SPADE.

It is important to note that SPADE is not meant to replace existing query optimizers. In this chapter we identify and quantify the effect of RDMA-specific choices on the performance of OLTP workloads in FaRM, as an additional level of necessary optimizations. Similarly, SPADE might not be applicable to all distributed transaction systems, since not all of them offer the possible choices in SPADE. However, as such systems become popular, there are lessons to be learned for related future efforts.

The rest of the chapter is organized as follows. In Section 6.2 we present the architecture of SPADE and in Section 6.3 we describe how tuning is performed. We evaluate workloads tuned with SPADE in Section 6.4 and present some concluding remarks in Section 6.5.

## 6.2 SPADE

In this section, we first describe SPADE's architecture and how it fits in the execution model of an OLTP workload. We then present how the compilation of query plans into low-level transactions works, how SPADE tunes the physical design, as well as the models it uses and how these are fed with data.

### 6.2.1 Architecture overview

Figure 6.2 presents SPADE's architecture. An application consists of application code and stored procedures. Stored procedures are written in a SQL dialect and compiled by a SQL compiler offline. In our implementation, we used the PostgreSQL compiler to produce physical

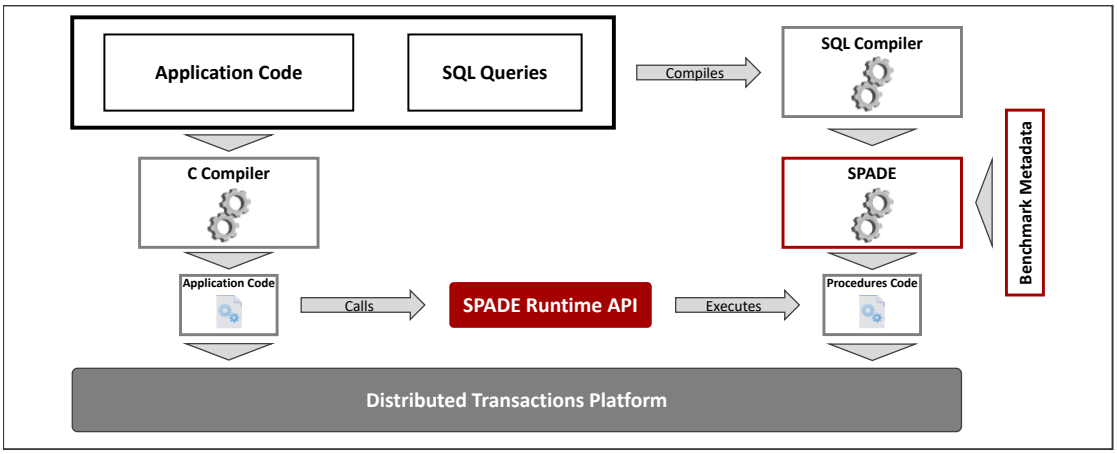



Figure 6.2 – The architecture of SPADE.

execution plans of the workloads (see Section 6.4.1). SPADE then analyzes the physical plans, tunes the indexes used based on the workload queries, and makes per-procedure choices regarding remote accesses. We leave support for online tuning as future work. The stored procedures are then compiled into low-level code that utilizes the distributed transactional platform's API to access data and indexes, similarly to the in-memory component of SQL Server [43]. The stored procedures are uploaded to an RDMA cluster together with the application. During execution, the application uses the SPADE runtime API to bind arguments and execute the stored procedures of the workload, as well as to retrieve the results of the queries.

Queries are compiled in two steps. Physical query plans, which describe the operators that need to be executed, the type of joins to use, and similar, are produced first using a SQL compiler. We use existing techniques for query compilation and do not impose any specific requirements on the physical plan. SPADE takes as input the query execution plan produced by an optimizer. With these plans as input, SPADE produces low-level code that implements the stored procedures, tuning their physical design, using the API offered by the distributed transactional platform.

To tune the physical design of queries, SPADE models each operation (e.g., lookups, updates, etc.) as a set of low-level distributed operations (e.g., remote reads and writes). To quantify the performance of these operations, SPADE executes a set of microbenchmarks on the target cluster. These include single operations (lookups, updates, insertions and deletions) executed both through a primary and a secondary index, for different cluster and row sizes. These microbenchmarks need to be run only once. SPADE executes three different scenarios for each operation: a) the operation accessing local data, b) the operation accessing remote data through RDMA, and c) the operation executing locally at a remote machine through an RPC. The results of these benchmarks are used by SPADE. We detail how SPADE uses this metadata for each tuning decision it automates in the following.

```
void get_subscriber_data(int s_id) {
  auto transaction_object =
    new SQLTransaction("GET_SUBSCRIBER_DATA");

  transaction_object->AddArgument(0, s_id);
  auto results = transaction_object->Execute();

  if (results->status == TxStatus::Committed)
  {
    // Application code to use the results
  }
}
```

Listing 6.1 – TATP's *GET_SUBSCRIBER_DATA* invoked in an application.

We present an example of how the *GET_SUBSCRIBER_DATA* query presented in Listing 2.1 can be invoked as part of an application in Listing 6.1. The code creates an object based on the name of the stored procedure (a name given by the developer) and invokes the *Execute* method. The application can check whether the transaction committed, and use the results of the query (in this case the subscriber data). If the SQL transaction has aborted, the application can retry the same transaction, based on whether the abort was due to contention or lack of data (not shown here).

## 6.3 Tuning

In this section, we discuss the tuning decisions we consider in SPADE. For each one we first present the possible choices and some intuition on how these affect performance. We then show examples of how these different options impact performance based on our microbenchmarks, and how we use this information in SPADE to tune a workload. We distinguish two categories of design choices, which we describe in the following subsections. We assume a row-store format, which is the prevalent format for OLTP workloads.

### 6.3.1 Index tuning

Indexes are an important part of databases and can significantly speed up access to data. SPADE supports two types of indexes: hash indexes and B-Tree indexes. Modern distributed transaction systems provide at least one of the two. Tuning these indexes can optimize the size and the number of network accesses. Typically a trade-off exists–increasing the size of accesses reduces the number of accesses necessary, but at the same time it reduces the request rate the hardware can perform. In SPADE we identified two opportunities for tuning indexes. The first choice is between storing records inside the data structures (e.g., as the value of a hashtable implementing a hash index) or storing only a pointer to the record in the index data structure. We refer to the first as *inlined indexes* and to the second as *non-inlined* indexes. The second choice is on the neighbourhood sizes for hash indexes and node sizes for B-Trees.

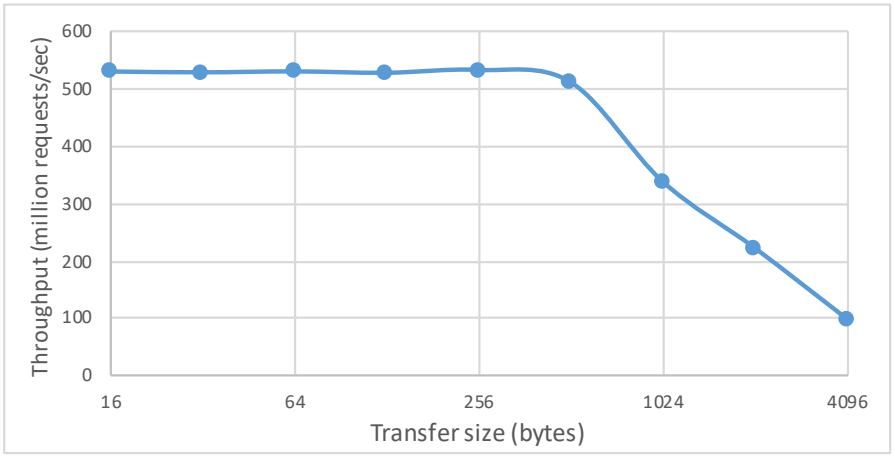
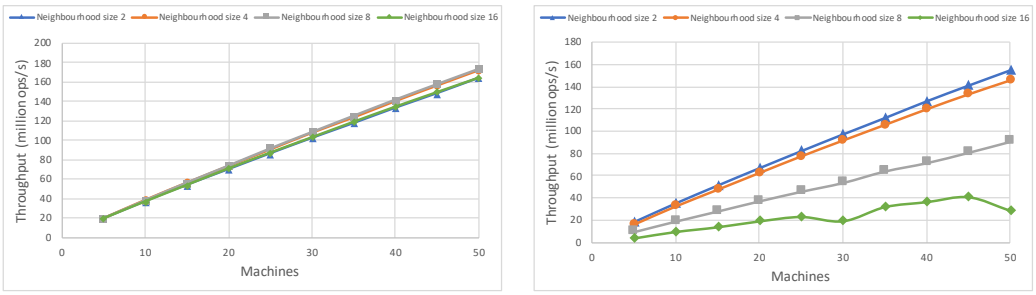



Figure 6.3 – Performance of random RDMA reads.

We use a hash index as an example of performance impact, but similar observations hold for B-Trees. Hash indexes in FaRM are implemented using *chained associative hopscotch hashing* [52]. Each bucket in the hashtable has an overflow chain, which stores key-value pairs that do not fit into the bucket. Lookups use one-sided RDMA reads to access a bucket, as well as to fetch the overflow chain blocks. A small bucket will typically lead to more RDMA accesses for the overflow chain. In contrast, a large bucket will result in fewer RDMA accesses, but its performance might suffer because of the large RDMA transfers. In Figure 6.3 we present the throughput of RDMA requests for different transfer sizes on a cluster of 50 machines (our setup is described in Section 6.4.2). For transfers of up to 256 bytes, the request rate remains constant. However, for sizes above 512 bytes the request rate degrades significantly. The specific numbers change based on the network hardware available.

Storing the records in a hashtable (and similarly in a B-Tree) raises the question of inlining. When the data is inlined, lookups can be as fast as a single one-sided RDMA. A larger neighbourhood size for an inlined table minimizes the number of RDMA transfers. At the same time, however, it results in larger RDMA transfers, which as we showed can reduce performance. There is a trade-off between keeping the neighbourhood small and reading as many records per hash index lookup as possible. Figure 6.4 presents the performance of primary index lookups using one-sided RDMA on two tables, with rows of 16 and 256 bytes. Both tables are inlined. For the smaller row size, a large neighbourhood of 8 rows per hash bucket achieves the highest throughput, while for larger records the best performance is achieved with a neighbourhood size of 2, and the throughput gains over choosing a neighbourhood of size 8 is 70% on 50 machines. Inlining does not create multiple copies of the data: if the data resides in the primary index, it is stored only there, and any secondary index merely holds the key of the data in primary one. This is possible in non-traditional systems, such as the ones we study, where there is no notion of "tables" and data can be stored anywhere.



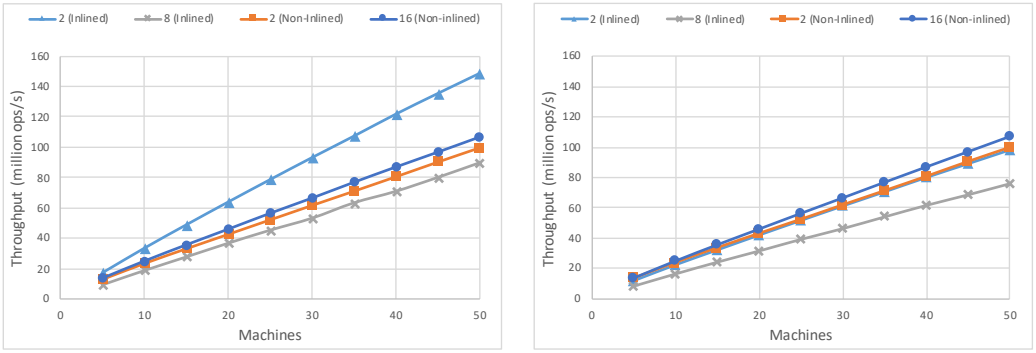

(a) 90% primary index - 10% secondary index lookups.

(b) 10% primary index - 90% secondary index lookups.

Figure 6.4 – Performance of inlined lookups.

Alternatively, the data could be stored outside of the index, with only a pointer stored. Pointers are typically smaller than rows: in FaRM they are 8 bytes. Holding the data outside indexes typically incurs extra memory accesses. However, records are not only accessed through a primary index (i.e. through the primary key of a relational table), and operations are not always read-only. For example, records are often accessed through secondary indexes, with the results of a lookup corresponding to more than one records. Retrieving the results of a secondary index lookup involves either looking up the records in the primary index (if inlined), or reading them directly from the memory (if not inlined), which is more efficient.

Figure 6.5 compares the performance of a read-only workload with two different transaction mixes: (1) 90% accesses through the primary index and 10% through a secondary one, and (2) 90% accesses through a secondary index and 10% through the primary. Rows are 256 bytes. The best configuration for the first transaction mix is to inline data and use a neighbourhood size of 2. Using the same configuration parameters for the second transaction mix yields up to 12.4% lower throughput than the optimal one, which is to not inline the data and use a neighbourhood size of 16.




(a) 90% primary index - 10% secondary index lookups.

(b) 10% primary index - 90% secondary index lookups.

Figure 6.5 – Performance of 256-byte row lookup mix.

To choose a configuration for a given table and transaction mix, SPADE works as follows. After translating all the queries of a workload into low-level distributed operations, SPADE groups operations per relational table (based on the workload schema). For each table $T$, SPADE calculates the weighted average of both throughput and latency:

$$Throughput(T) = \sum_i r_i(T) * Thr_i(T)$$

$$Latency(T) = \sum_i r_i(T) * Lat_i(T)$$

Where $r_i(T)$ is the ratio of the low-level operation $i$ (e.g., lookup through a hash index, update through a secondary index, etc.) across all the operations on table $T$:

$$r_i(T) = \frac{N_i(T)}{\sum_j N_j(T)}$$

$N_i(T)$ is the number of occurrences of operation $i$ on table $T$. If metadata about the query mix in a workload is available (we use such metadata in our evaluation), $N_i(T)$ is adjusted by the ratio of each query to the total mix. SPADE then iterates over the parameter space, using the microbenchmark performance numbers for each operation – $Thr_i(T)$ and $Lat_i(T)$. SPADE then tries to achieve a goal, such as maximizing $Throughput(T)$ or minimizing $Latency(T)$. We leave the implementation of more elaborate heuristics in SPADE for future work.

### 6.3.2 Partitioning and remote access optimization

An important decision in scale-out systems is partitioning: ideally, data should be sharded in such a way that all transactions access only local data. With modern datacenter networks, partitioning is no longer the main performance factor but rather an optimization: accessing remote memory is fast, but still an order of magnitude slower than accessing local memory [174].

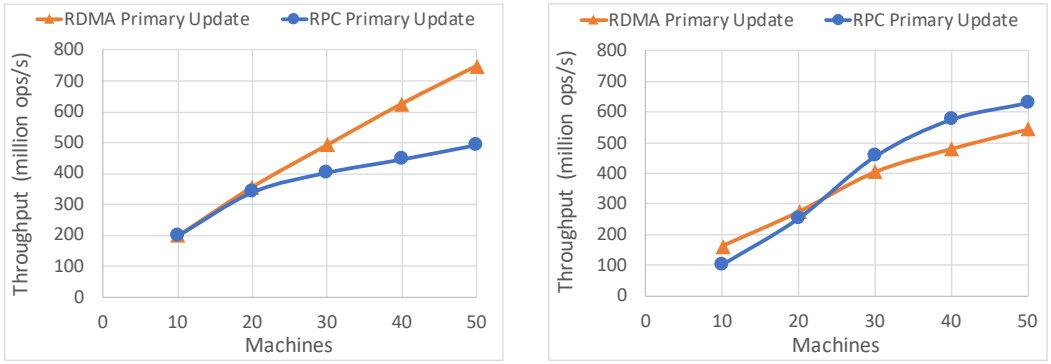
There is a large body of work on data partitioning [6, 7, 123, 127, 133, 168]. SPADE is not meant to replace these systems. However, using a shared-memory compiler and optimizer, we implement a simple partitioning scheme in SPADE. Before the tuning process, SPADE groups accesses to tables per transaction. SPADE then identifies stored procedure inputs that are used to access more than one tables, creating groups of tables that are accessed with the same primary key, or common parts of their primary keys. It also keeps these common keys that act as *links* in this grouping. After that, SPADE identifies intermediary outputs of a stored procedure used to access tables (e.g., foreign keys) and further groups tables. Finally, SPADE uses the *links* identified as the partitioning key. In FaRM, the distributed data structures used as indexes (the hashtable and B-Tree) support partitioning keys, which allows SPADE to collocate parts of tables that will be accessed together.

```
UPDATE Subscriber
SET bit_1 = <rnd>
WHERE s_id = <rnd subid>;
UPDATE Special_Facility
SET data_a = <rnd>
WHERE s_id = <s_id value subid>
AND sf_type = <rnd>;
```

Listing 6.2 – TATP's *UPDATE_SUBSCRIBER_DATA* written in SQL

For example, from Listing 6.2, SPADE can identify that records of the *Subscriber* and *Special_Facility* tables are accessed together based on the *s_id*. Thus, a grouping between the two is created, with *s_id* being the *link*. After all the tables are processed, SPADE partitions the *Subscriber* and *Special_Facility* tables with *s_id* as the partitioning key of the hash indexes used for both tables. Thus, accesses local to a specific subscriber, will always result in local accesses to the special facilities that correspond to that subscriber. We believe that more elaborate partitioning and collocation heuristics can be included in the SQL compiler and provide additional inputs to SPADE. We thus focus on tuning remote accesses for transactions.

In the general case, it is unavoidable for some workloads to access remote data. SPADE distinguishes two cases: a) a transaction refers to remote data, which is all on the same machine or b) a transaction refers to a combination of local and remote data residing on multiple machines. SPADE further optimizes for case (a). There are two possible ways to execute a transaction that only accesses data on a single remote machine. The first is to utilize RDMA to fetch the necessary data, invoking the remote protocol at the end of the transaction. The second approach is to do an RPC to the remote machine, have the transaction executed on that machine, and the results shipped to the invoking machine. The performance of even simple operations differs between these two choices. Figure 6.6 shows the performance of a single-row update for two different row sizes, 16 and 256 bytes. In these measurements, we include the cost of replication (we use a replication factor of 3, similarly to our evaluation setup). Persistence is also included, since we assume main memory that has power support,




(a) 16-byte rows.

(b) 256-byte rows.

Figure 6.6 – Performance of RDMA and RPC for a single-row update operation.

as described in 2.4 and implemented in FaRM. This example assumes that we are not updating the full row, but a single field in it, which is the case for many workloads. While for small row-sizes accessing the row through RDMA and invoking the distributed commit protocol achieves higher throughput, for the 256-byte rows sending an RPC and invoking the commit protocol at the machine where the data resides is 16% faster. The difference in performance only becomes bigger as the transactions to be executed become longer and touch more data.

SPADE optimizes remote transaction execution using a similar technique as before. Instead of grouping operations per relational table, SPADE examines low-level operations (including distributed data structure accesses) of a single workload transaction, possibly consisting of multiple queries. For each transaction $Tx$, it calculates the weighted average of both throughput and latency:

$$Throughput(Tx) = \sum_i r_i * Thr_i(Tx)$$

$$Latency(Tx) = \sum_i r_i * Lat_i(Tx)$$

Where $r_i^T$ is the ratio of the low-level operation $i$ (e.g., lookup through a hash index, update through a secondary index, etc.) over all the operations in transaction $Tx$:

$$r_i = \frac{N_i(Tx)}{\sum_j N_j(Tx)}$$

$N_i(Tx)$ denotes the number of occurrences of the operation $i$ in transaction $Tx$. SPADE then iterates over the two available choices (RDMA accesses or an RPC call for a transaction) and chooses the one that either maximizes $Throughput(Tx)$ or minimizes $Latency(Tx)$. If SPADE chooses an RPC for a transaction, it generates the code necessary for the RPC calls.

## 6.4 Evaluation

We used SPADE to implement two popular OLTP workloads: TATP and TPC-C. We first discuss TATP, establishing a baseline implementation and then showing how the different tuning decisions included into SPADE affect performance, what wrong choices would mean for both throughput and latency. Then we connect these results to the observations of Sections 6.3.1 and 6.3.2. We also present the final results for both workloads in terms of throughput, median and 99$^{th}$ latency when scaling out the workloads to a cluster of 50 machines. We focus on the performance of TATP because of the different types of queries it contains: TATP has both read-only and read-write transactions, as well as transactions that touch a varied amount of data. As such, a single configuration for all tables and indexes will be far from optimal. We use such examples throughout our evaluation.

### 6.4.1 Implementation

We implement SPADE as a standalone application written in F#. Our microbenchmarks are written in C++ and executed once for a FaRM cluster. We use PostgreSQL to generate plans for our workloads in XML format, which are then used as input to SPADE. SPADE produces executable code for the queries of a workload, which is invoked by applications using a C++ API we developed for SPADE. Both applications and compiled transaction code are uploaded to a FaRM cluster for execution. At runtime, the compiled code of a stored procedure invokes the FaRM API. Fault-tolerance and partition guarantees are the same as described in [53]. Additionally, due to the power-backed memory in a FaRM cluster, data is durable at all times. As a result, our system and execution model provides ACID transactions.

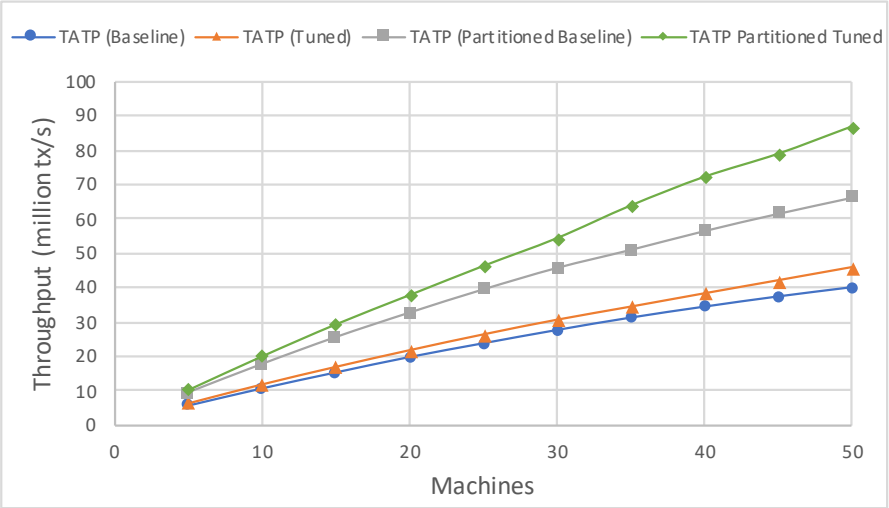
### 6.4.2 Experimental setup

We run our experiments on a cluster of 50 machines. Each machine has 256 GB of DRAM and two Intel E5-2650 CPUs (16 cores / 32 threads in total). Each machine has two Mellanox ConnectX-3 56Gbps Infiniband NICs, one shared by the threads of each socket. The machines are connected to a single Mellanox SX6512 switch with full bisection bandwidth. The cluster runs Windows Server 2016. FaRM is configured with 3-way replication. All experiments are run for 60 seconds (excluding warm-up) and 3 iterations. In the graphs presented we report the average of the 3 iterations (we do not notice significant deviations).

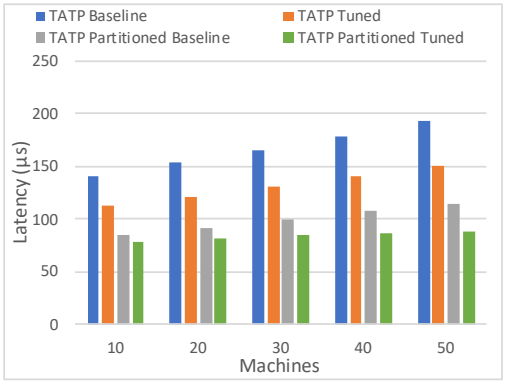
### 6.4.3 TATP

**Overview.** The Telecommunications Application Transaction Processing (TATP) [120] is a relational benchmark, simulating a typical Home Location Register (HLR) database in a mobile phone network. The benchmark consists of a set of seven pre-defined transactions that query, update and insert data in 4 tables, following a fixed probability for each transaction (referred to as *transaction mix*). TATP is read dominated, with 80% of the transactions being read-only, and only 20% being read-write transactions. It also involves small transactions, with the rows read/written being between 8 and 72 bytes. Throughout our experiments, we use 50 million subscribers per machine (250 million subscribers for a cluster of size 5 and 2.5 billion subscribers for a cluster of size 50), for a total size of 1TB of data on 50 machines.

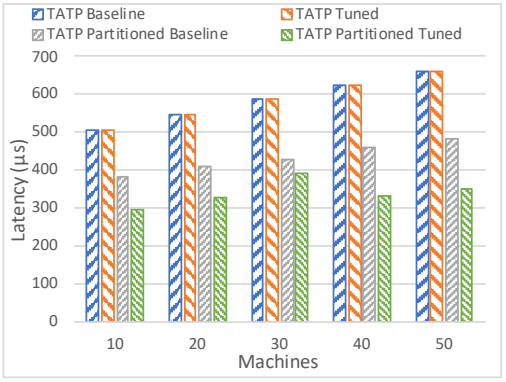
In all our experiments, we scale TATP to 50 machines. To showcase the effect of tuning decisions, we write a baseline implementation of TATP and then use SPADE to tune the different parts of the workload. We repeat the experiments and report the difference in throughput and latency (median and $99^{th}$ percentile). Figure 6.7 shows the throughput and latency results of our experiments.




(a) Throughput.




(b) Median latency.




(c) 99$^{th}$ latency.

Figure 6.7 – Performance of all TATP versions.

**Baseline.** We begin by implementing a first version of TATP, using the transactions as described in the specification. Our application invokes a random number generator, and based on the workload mix described in the TATP specification, executes the different transactions.

For this baseline, the data is not partitioned, with tables distributed across the cluster in a round-robin fashion. Remote accesses use one-sided RDMA reads and writes. All indexes are configured with the same parameters, which favor small row sizes, as most of the TATP tables have small rows. We refer to this version as *TATP Baseline*. *TATP Baseline* achieves a throughput of 39.9 million committed transactions per second on 50 machines, with a median latency of 193 microseconds and a 99$^{th}$ percentile latency of 660 microseconds.

**Index tuning.** We use SPADE to tune the indexes used in TATP, as described in Section 6.3.1. We refer to this version as *TATP Tuned* in the remainder of this section. We use hash indexes for the primary keys of the TATP tables. The 4 tables of TATP have different characteristics, both in terms of row sizes (ranging from 8 to 72 bytes in our implementation), but also significant

differences in accesses. For example, the *Access_Info* table has a row size of 10 bytes, and is only read by the queries. In contrast, the *Call_Forwarding* rows are 20 bytes and the rows are read, inserted, and deleted. Correctly configuring each index improves throughput by 10%-15% and latency by 19%-22% across different cluster sizes.

**Partitioning.** Next, we partition the TATP workload. SPADE identifies that most accesses are done based on a specific subscriber (the *s_id* of the subscriber table). Thus, the tables are partitioned in a way that the rows of all the tables that refer to a subscriber are stored on the same machine. Transactions access local data when invoked on the machine that stores the subscriber affected. For the rest of our experiments, we hardcode 10% of the transactions to reference subscribers on a different machine, to ensure that there are remote accesses across the cluster. Initially, all remote accesses are executed using RPCs: a machine first looks up the machine storing the data, and then does an RPC to that machine to execute the transaction and return the result. We refer to this implementation as *TATP Partitioned Baseline*.

In Figure 6.7, we show the improvement over *TATP Tuned*. The results are intuitive: By avoiding remote accesses, the majority of transactions is executed locally, avoiding communication through RPCs. In our implementation, when a transaction accesses remote data, the machine processing the transaction does an RPC to the machine holding all the data necessary to execute the transaction, so the accesses to memory are local (except for the shipping of results). The improvement in throughput is between 45% and 50%, while latency is improved between 23% and 25% for the median and between 22% and 28% for the 99$^{th}$ percentile latency. The improvement is significant. However, it is comparable to that between *TATP Baseline* and *TATP Tuned*. In the past crossing partitions and accessing remote data used to be prohitibitively expensive. As this experiment shows, with modern network interconnects, the effect of remote accesses becomes smaller, and partitioning can be viewed as an optimization, rather than a requirement.

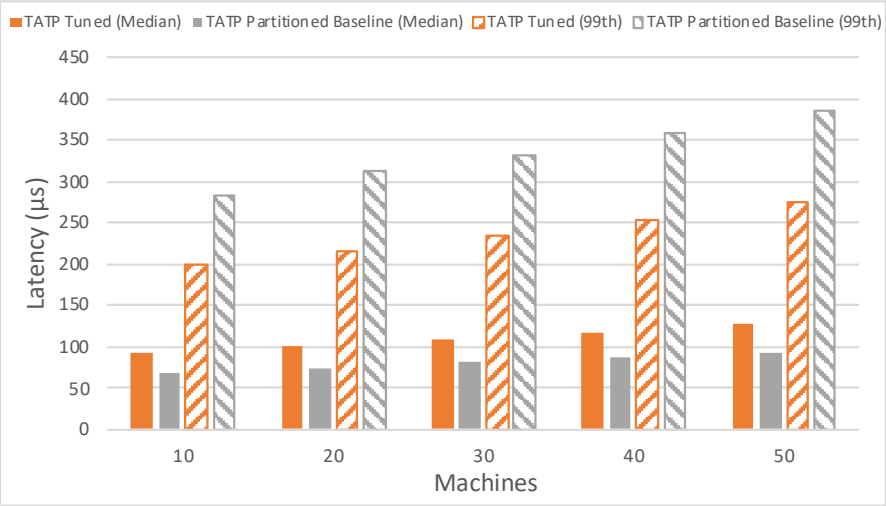



Figure 6.8 – TATP's *GET_ACCESS_DATA* transaction latency after partitioning.

```
SELECT data1, data2, data3, data4
FROM Access_Info
WHERE s_id = <rnd> AND ai_type = <rnd>;
```

Listing 6.3 – TATP's "GET_ACCESS_DATA" in SQL

Different transactions benefit from SPADE's optimizations to different extents. For instance, for the *GET_ACCESS_DATA* transaction (the SQL for this transaction is shown in Listing 6.3) the median latency is 25% lower, but the 99$^{th}$ percentile latency is increased by 40-53% (Figure 6.8). This is because the query is read-only, and accesses a single row of the table. Doing an RPC when accessing remote data adds an extra overhead to the transaction: the data is shipped to the calling machine and by using an RPC we additionally involve the remote CPU. However this is the case only because of the size of the returned data (10 bytes) and the length of the transaction itself (a single lookup). Because of the partitioned workload, in *TATP Partitioned Baseline* version, 90% of the invocations of this transaction will access local data, lowering the median latency, while the 10% of the cases where remote accesses are necessary will be slower than before (our *TATP tuned* version uses one-sided RDMA operations for all remote accesses). This transaction is a perfect example of a physical design choice that is tied to the characteristics of the data and the underlying hardware, but also one that would require extensive experimentation to identify. A uniform configuration for all the remote accesses clearly is not optimal for TATP.

**Remote access tuning.** Finally, we enable the cost models of SPADE which decide on whether an RPC should be preferred over a direct access to data (we refer to this implementation as *TATP Partitioned Tuned*. Throughput is improved between 9% and 30% and latency between 5% and 22%. This is a direct result of addressing all the cases such as *GET_ACCESS_DATA*, which benefit from directly accessing remote data over message passing. Figure 6.9 shows the

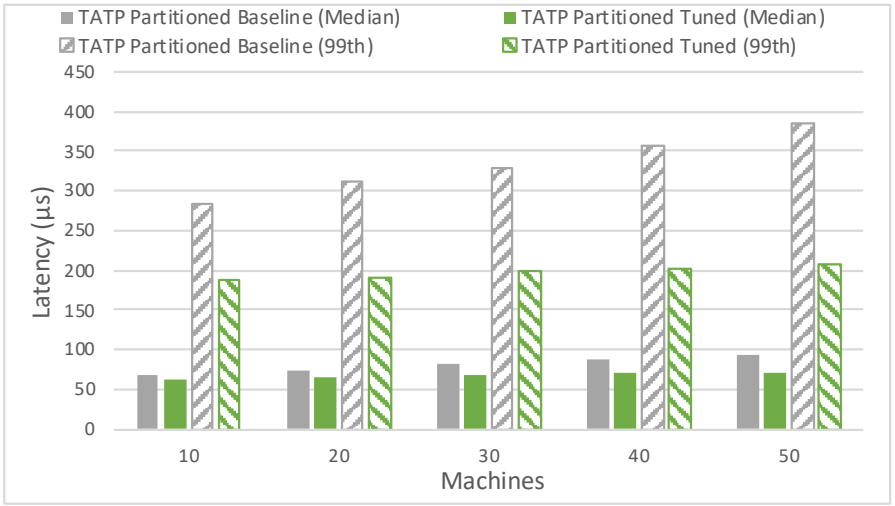
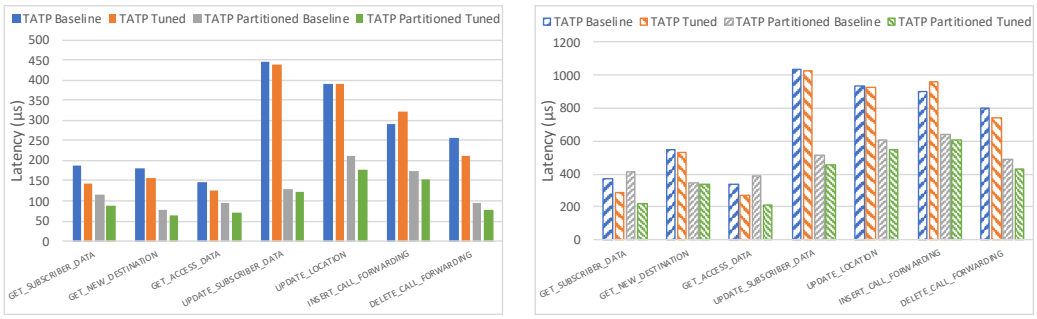



Figure 6.9 – TATP's *GET_ACCESS_DATA* transaction latency with tuned remote accesses.




(a) Median latency.

(b) 99$^{th}$ latency.

Figure 6.10 – Overview of the impact of different physical design choices to TATP operation latency.

improvement for this particular query (introduced before). Optimizing parts of the workload to use RDMA improves median latency between 5% and 23% and tail latency by up to 45%. Doing so manually is far from trivial. Using a cost model like the one we use in SPADE is not only easier, but also more accurate, since it can calculate the optimal setup for a set a stored procedures and a specific transaction mix.

**Effect on individual operations.** Figure 6.10 presents the median and 99$^{th}$ percentile latencies of all 7 stored procedures in TATP across the four versions presented on 50 machines. For 4 of the 7 transactions, each set of tuning decisions improves median latency, lowering it by as much as 70%. For both *UPDATE_SUBSCRIBER_DATA* and *UPDATE_LOCATION*, the differences when optimizing indexes (*TATP Tuned)* are small: for the Subscriber table, SPADE chooses to store data outside of the primary hash index. Thus, both the primary index and the secondary index on *sub_nbr* store pointers to the objects. As a result, lookups through the secondary index are significantly faster, but updates become slower, since they require more operations (as opposed to inlined data). However, the total effect on the workload is positive. If the transaction mix was different, SPADE might choose to inline the *Subscriber* table, in order to achieve an overall better performance.

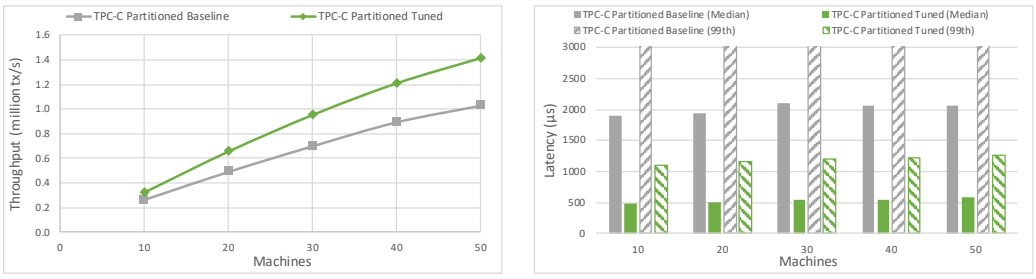
Similarly, each set of tuning decisions improves the 99$^{th}$ percentile latency of transactions, with two exceptions. For TATP, partitioning data translates to co-locating rows of the different tables based on the subscriber they refer to (*s_id*). For *GET_SUBSCRIBER_DATA* and *GET_ACCESS_DATA*, using only RPCs for transactions accessing remote data increases 99$^{th}$ percentile latency by 40.9% and 40.1% respectively. Both transactions are read-only, touching small amounts of data. Doing an RPC and receiving the results translates to higher latency than directly reading the necessary data using one-sided RDMA reads. The transfer latency for the data will be the same, however, using message-passing will additionally involve the remote CPU, which will have to process the request and send the response. For both transactions, when switching to RDMA reads for remote accesses, median latency improves by 23%, and at the same time tail latency is lowered by 45%.

### 6.4.4 TPC-C

**Overview.** The Transaction Processing Performance Council Benchmark C (TPC-C) [159] is a well-known and long-studied OLTP benchmark. TPC-C emulates the activity of a wholesale supplier, with tables that contain data on stock, customers, new orders, and more. The workload consists of five pre-defined transactions of different types and complexity, which can access up to hundreds of rows. In contrast to TATP, TPC-C is write-intensive, with 35.8% of transactions modifying data. Throughout our experiments, we use 120 warehouses per machine, for a total of 6000 warehouses on 50 machines.

We scale TPC-C to 50 machines. Since TPC-C is partitionable by design (the granularity of partitioning is a warehouse), we do not use SPADE to partition the workload, and focus on remote transaction processing, using either RPCs or RDMA accesses. We collect statistics for all the transactions of the workload and report throughput and latency for "new order" transactions only.

**Effect of remote accesses.** Figure 6.11 depicts the effect of optimizing remote accesses for TPC-C. In TPC-C warehouses are the granularity of partitioning: 90% the queries reference data that corresponds to a local warehouse, as such they result in local data accesses. Thus, we begin our tuning process from the *TPC-C Partitioned Baseline* implementation, which uses the same configuration for all indexes and RDMA accesses for all remote data, and then invokes the distributed commit protocol. Since we are using FaRM for our implementation, during commit, read values are validated to have not changed, and modified values are shipped to the machine storing the data (and then replicated to backups). The second version, named *TPC-C Partitioned Tuned*, is the result of using SPADE to tune indexes and remote accesses. SPADE decides that RPCs are better for all the transactions accessing remote data in TPC-C. Using RDMA for transactions that reference remote data yields throughput that is lower by up to 37.7%, while median and tail latency are up to 298% and 395% higher respectively. All the queries executed in TPC-C involve numerous data accesses. Using RDMA decreases throughput and increases latency. Additionally, due to longer transactions,




(a) Throughput.

(b) Median and 99$^{th}$ latency.

Figure 6.11 – Performance of *TPC-C* and remote access tuning.

throughput is further lowered, due to the closed-loop architecture of the TPC-C benchmark. Adding concurrent transactions improves throughput slightly, but at the cost of higher latency.

## 6.5 Conclusions

This chapter introduced SPADE, the first physical design tuner for OLTP workloads on modern RDMA clusters. SPADE automates choices regarding the tuning of indexes, the partitioning of data, and remote accesses, shipping data, or the computation through RPCs for different parts of the workload. SPADE achieves this by analyzing the physical plan of a workload, and using hardware and network performance characteristics gathered through microbenchmarks. Our evaluation of workloads tuned using SPADE showed that these choices have a significant impact on both throughput and latency, improving them by up to 117% and 75% respectively.

# Part IV

# Concluding Remarks

# 7 Conclusions and Future Directions

Synchronization is an important part of modern software, allowing for concurrent executions that improve performance, while ensuring correctness. However, with modern hardware platforms containing tens of processing cores, large main memory capacities, and fast network interconnects in the datacenter, synchronization can become a significant bottleneck for the scalability of applications. This dissertation studied the scalability of synchronization on modern hardware, focusing on the shared-memory setting for applications. In this chapter, we summarize the main contributions of our work and present some directions for future work.

## 7.1 Understanding the Scalability of Shared-Memory Applications

In this dissertation, we argued that understanding the scalability of applications during development can be tedious. Typically, development takes place on small desktop machines, and not every developer has dedicated access to large servers. Creating detailed models and simulations of applications and hardware platforms is not only tedious and time-consuming, but also error-prone.

In Chapter 4 we presented ESTIMA, a practical tool that extrapolates the scalability of in-memory applications. ESTIMA uses a combination of hardware performance counters, as well as software-measured performance metrics. This allows for wider applicability, since ESTIMA does not require extensive instrumentation or access to the application source code.

The key contribution of our work can be summarized as the use of performance counters, a tool typically used for bottleneck identification, to extrapolate the scalability of applications. Moreover, with ESTIMA, we also introduce how one can use software-sourced performance metrics to understand the effect of synchronization on the scalability of an application.

In our work, we included software stalls from threading libraries such as the `pthread` library, as well as from the STM runtime we used. Although HTM implementations were already present at that moment, they did not offer enough performance counter metrics for us to include support. As HTM becomes more popular, we believe that more fine-grain performance

analysis tools and support will be necessary, in order to understand potential bottlenecks in implementations and guide developers towards fixing them.

Another issue that we faced during this work, was a lack of support for monitoring of multiple performance counters in parallel. This leads to both performance penalties, as well as inaccuracies in the measured values. We believe that as hardware platforms become more and more complex, understanding how to build efficient software will require better support from the hardware in understanding the behavior of an application on a specific hardware platform.

## 7.2 Improving the Scalability of Shared-Memory Applications

In the second part of this dissertation, we focused on improving two common synchronization primitives: locking and transactions.

Locking is one of the most straightforward approaches to synchronizing access to data. In Chapter 5 we claimed that using locks effectively requires a significant amount of effort, and can lead to both correctness and performance issues when implemented incorrectly. We introduced GLS, a runtime that abstracts all of these design and implementation choices from the developer, allowing for easier development of multi-threaded applications. Our contribution can be summarized as the introduction of a different locking paradigm, where locks are managed by a runtime, while the developer only defines the needs of the application.

Although our work made a significant step towards better locking performance and usability, the performance of individual lock algorithms and implementations is still an open problem. More specifically, as concurrent work has shown [66], locking algorithms and their implementations across different hardware platforms can significantly affect their performance. Thus, it is important to find ways to ensure portable performance for the widely used locking algorithms. We expect the need for such work to only increase as hardware becomes more diverse in the near future.

Finally, we studied the current landscape in datacenter distributed transactions. We identified that achieving performance for OLTP workloads requires significant manual effort. In Chapter 6 we quantified the effect of this tuning on the performance of OLTP workloads and presented SPADE, a tuner for OLTP workloads on modern RDMA clusters. SPADE automates decisions on storage parameters, index tuning and remote data accesses, on top of traditional optimizations for workloads performed by query optimizers. To do so, SPADE uses performance characteristics of the hardware and network, and combines them with workload and dataset information, to automate this tuning.

Our work did a first step towards offering the power of distributed transactions to widely used workloads, removing the need for extensive manual effort and detailed knowledge of the hardware and workload from the developers. However, we have barely scratched the surface of an interesting area of research. A question arising from our work is how we can use

similar techniques to improve On-Line Analytical Processing (OLAP) workloads on similar datacenter environments, taking into account the differences in the characteristics between OLTP and OLAP workloads. Moreover, an interesting direction is whether we can combine the two workload characteristics, and how existing work [9] will need to be adapted to better take advantage of modern hardware.

# Bibliography


[1] Pthread Mutex Lock. https://sourceware.org/git/?p=glibc.git;a=blob_plain;f=nptl/pthread_mutex_lock.c;hb=HEAD. Accessed: 2018-06-20. [Page 19]

[2] C99 Standard. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf, 2007. [Pages 69 and 72]

[3] Information technology Portable Operating System Interface (POSIX) Base Specifications, Issue 7. *IEEE/ISO/IEC 9945*, 2009. [Page 72]

[4] Naum I Achieser. *Theory of approximation*. Dover Publications, 1992. [Page 31]

[5] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database tuning advisor for microsoft SQL server 2005. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 1110–1121, 2004. [Page 21]

[6] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 359–370, 2004. doi: 10.1145/1007568.1007609. URL http://doi.acm.org/10.1145/1007568.1007609. [Page 86]

[7] Sanjay Agrawal, Eric Chu, and Vivek R. Narasayya. Automatic physical design tuning: workload as a sequence. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 683–694, 2006. doi: 10.1145/1142473.1142549. URL http://doi.acm.org/10.1145/1142473.1142549. [Pages 21 and 86]

[8] Marcos K. Aguilera, Joshua B. Leners, and Michael Walfish. Yesquel: scalable sql storage for web applications. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 245–262, 2015. doi: 10.1145/2815400.2815413. URL http://doi.acm.org/10.1145/2815400.2815413. [Pages 2, 15, and 79]

[9] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 169–180, 2001. URL http://www.vldb.org/conf/2001/P169.pdf. [Page 101]

[10] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1009–1024, 2017. doi: 10.1145/3035918.3064029. URL http://doi.acm.org/10.1145/3035918.3064029. [Page 21]

[11] Alaa R. Alameldeen and David A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, 2006. doi: 10.1109/MM.2006.73. URL http://dx.doi.org/10.1109/MM.2006.73. [Page 11]

[12] AMD. BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors, 2010. [Pages 27 and 33]

[13] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan 1990. ISSN 1045-9219. doi: 10.1109/71.80120. [Pages 12 and 19]

[14] Jelena Antic, Georgios Chatzopoulos, Rachid Guerraoui, and Vasileios Trigonakis. Locking made easy. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*, page 20, 2016. URL http://dl.acm.org/citation.cfm?id=2988357. [Page iii]

[15] Cosmin Arad, Jim Dowling, and Seif Haridi. Message-passing concurrency for scalable, stateful, reconfigurable middleware. In *Middleware 2012 - ACM/IFIP/USENIX 13th International Middleware Conference, Montreal, QC, Canada, December 3-7, 2012. Proceedings*, pages 208–228, 2012. doi: 10.1007/978-3-642-35170-9_11. URL https://doi.org/10.1007/978-3-642-35170-9_11. [Page 20]

[16] David F. Bacon, Ravi B. Konuru, Chet Murthy, and Mauricio J. Serrano. Thin locks: Featherweight synchronization for java. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 258–268, 1998. doi: 10.1145/277650.277734. URL http://doi.acm.org/10.1145/277650.277734. [Page 19]

[17] Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis R. de Supinski, and Martin Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008, Island of Kos, Greece, June 7-12, 2008*, pages 368–377. ACM, 2008. doi: 10.1145/1375527.1375580. URL http://doi.acm.org/10.1145/1375527.1375580. [Pages 11, 17, and 25]

[18] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. Rack-scale in-memory join processing using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1463–1475, 2015. doi: 10.1145/2723372.2750547. URL http://doi.acm.org/10.1145/2723372.2750547. [Page 15]

[19] Andrew Baumann, Paul Barham, Pierre-Évariste Dagand, Timothy L. Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 29–44, 2009. doi: 10.1145/1629575.1629579. URL http://doi.acm.org/10.1145/1629575.1629579. [Pages 2 and 20]

[20] Peter Belknap, Benoît Dageville, Karl Dias, and Khaled Yagoub. Self-tuning for SQL performance in oracle database 11g. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 1694–1700, 2009. doi: 10.1109/ICDE.2009.165. URL https://doi.org/10.1109/ICDE.2009.165. [Page 21]

[21] Arnd Bergmann. BKL: That's All, Folks. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=4ba8216cd90560bc402f52076f64d8546e8aefcb, 2011. Accessed: 2018-06-20. [Pages 14 and 55]

[22] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *17th International Conference on Parallel Architecture and Compilation Techniques (PACT 2008), Toronto, Ontario, Canada, October 25-29, 2008*, pages 72–81. ACM, 2008. doi: 10.1145/1454115.1454128. URL http://doi.acm.org/10.1145/1454115.1454128. [Pages 27 and 37]

[23] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, 2011. doi: 10.1145/1941487.1941507. URL http://doi.acm.org/10.1145/1941487.1941507. [Page 1]

[24] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. Frans Kaashoek, Robert Tappan Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 43–57, 2008. URL http://www.usenix.org/events/osdi08/tech/full_papers/boyd-wickizer/boyd-wickizer.pdf. [Page 20]

[25] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Tappan Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 1–16, 2010. URL http://www.usenix.org/events/osdi10/tech/full_papers/Boyd-Wickizer.pdf. [Page 20]

[26] Davidlohr Bueso and Scott Norton. An overview of kernel lock improvements. *LinuxCon North America, Hewlett-Packard, Chicago*, 2014. [Pages 20 and 58]

[27] Laura Carrington, Allan Snavely, and Nicole Wolter. A performance prediction framework for scientific applications. *Future Generation Comp. Syst.*, 22(3):336–346, 2006. doi: 10.1016/j.future.2004.11.019. URL http://dx.doi.org/10.1016/j.future.2004.11.019. [Pages 18 and 25]

[28] Milind Chabbi and John M. Mellor-Crummey. Contention-conscious, locality-preserving locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 22:1–22:14, 2016. doi: 10.1145/2851141.2851166. URL http://doi.acm.org/10.1145/2851141.2851166. [Page 19]

[29] Georgios Chatzopoulos, Aleksandar Dragojevic, and Rachid Guerraoui. ESTIMA: extrapolating scalability of in-memory applications. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 27:1–27:11, 2016. doi: 10.1145/2851141.2851159. URL http://doi.acm.org/10.1145/2851141.2851159. [Page iii]

[30] Georgios Chatzopoulos, Aleksandar Dragojevic, and Rachid Guerraoui. ESTIMA: extrapolating scalability of in-memory applications. *TOPC*, 4(2):10:1–10:28, 2017. doi: 10.1145/3108137. URL http://doi.acm.org/10.1145/3108137. [Page iii]

[31] Georgios Chatzopoulos, Rachid Guerraoui, Tim Harris, and Vasileios Trigonakis. Abstracting multi-core topologies with MCTOP. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 544–559, 2017. doi: 10.1145/3064176.3064194. URL http://doi.acm.org/10.1145/3064176.3064194. [Page iii]

[32] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 26:1–26:17, 2016. doi: 10.1145/2901318.2901349. URL http://doi.acm.org/10.1145/2901318.2901349. [Pages 2, 15, 21, and 79]

[33] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001*, pages 73–88, 2001. doi: 10.1145/502034.502042. URL http://doi.acm.org/10.1145/502034.502042. [Pages 2, 14, and 55]

[34] Cristian Coarfa, John M. Mellor-Crummey, Nathan Froyd, and Yuri Dotsenko. Scalability analysis of SPMD codes using expectations. In *Proceedings of the 21th Annual International Conference on Supercomputing, ICS 2007, Seattle, Washington, USA, June 17-21, 2007*, pages 13–22. ACM, 2007. doi: 10.1145/1274971.1274976. URL http://doi.acm.org/10.1145/1274971.1274976. [Page 17]

[35] Douglas Comer. The ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979. doi: 10.1145/356770.356776. URL http://doi.acm.org/10.1145/356770.356776. [Page 15]

[36] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the AMD opteron processor. *IEEE Micro*, 30(2):16–29, 2010. doi: 10.1109/MM.2010.31. URL https://doi.org/10.1109/MM.2010.31. [Page 10]

[37] Travis S. Craig. Building FIFO and Priority-queuing Spin Locks from Atomic Swap. Technical report, Technical Report TR 93-02-02, University of Washington, 02 1993.(ftp tr/1993/02/UW-CSE-93-02-02. PS. Z from cs. washington. edu), 1993. [Pages 12, 19, and 57]

[38] Mark E. Crovella and Thomas J. LeBlanc. Parallel performance prediction using lost cycles analysis. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, Supercomputing '94, pages 600–609, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-6605-6. [Page 17]

[39] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010. URL http://www.comp.nus.edu.sg/~vldb2010/proceedings/files/papers/R04.pdf. [Pages 20 and 22]

[40] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 33–48, 2013. doi: 10.1145/2517349.2522714. URL http://doi.acm.org/10.1145/2517349.2522714. [Pages 14, 19, 55, 56, and 59]

[41] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 631–644,

2015. doi: 10.1145/2694344.2694359. URL http://doi.acm.org/10.1145/2694344.2694359. [Page 67]

[42] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974. ISSN 0018-9200. doi: 10.1109/JSSC.1974.1050511. [Page 1]

[43] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1243–1254. ACM, 2013. doi: 10.1145/2463676.2463710. URL http://doi.acm.org/10.1145/2463676.2463710. [Pages 25 and 82]

[44] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. Automatic performance diagnosis and tuning in oracle. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 84–94, 2005. URL http://cidrdb.org/cidr2005/papers/P07.pdf. [Page 21]

[45] Dave Dice, Maurice Herlihy, Doug Lea, Yossi Lev, Victor Luchangco, Wayne Mesard, Mark Moir, Kevin Moore, and Dan Nussbaum. Applications of the adaptive transactional memory test platform. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, pages 1–10, 2008. [Page 20]

[46] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: a general technique for designing NUMA locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 247–256, 2012. doi: 10.1145/2145816.2145848. URL http://doi.acm.org/10.1145/2145816.2145848. [Pages 19 and 61]

[47] Pedro C. Diniz and Martin C. Rinard. Eliminating synchronization overhead in automatically parallelized programs using dynamic feedback. *ACM Trans. Comput. Syst.*, 17 (2):89–132, 1999. doi: 10.1145/312203.312210. URL http://doi.acm.org/10.1145/312203.312210. [Page 19]

[48] Siying Dong. Reducing Lock Contention in RocksDB. https://rocksdb.org/blog/2014/05/14/lock.html, 2014. Accessed: 2018-06-20. [Page 20]

[49] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 155–165. ACM, 2009. doi: 10.1145/1542476.1542494. URL http://doi.acm.org/10.1145/1542476.1542494. [Pages 3, 33, and 35]

[50] Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Commun. ACM*, 54(4):70–77, 2011. doi: 10.1145/1924421.1924440. URL http://doi.acm.org/10.1145/1924421.1924440. [Pages 35 and 37]

[51] Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Commun. ACM*, 54(4):70–77, 2011. doi: 10.1145/1924421.1924440. URL http://doi.acm.org/10.1145/1924421.1924440. [Page 20]

[52] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 401–414, 2014. URL https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevic. [Pages 2, 4, 5, 15, 21, 79, 80, and 84]

[53] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 54–70, 2015. doi: 10.1145/2815400.2815425. URL http://doi.acm.org/10.1145/2815400.2815425. [Pages 4, 5, 14, 15, 21, 79, 80, and 89]

[54] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *PVLDB*, 2(1):1246–1257, 2009. doi: 10.14778/1687627.1687767. URL http://www.vldb.org/pvldb/2/vldb09-193.pdf. [Page 21]

[55] Dawson R. Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 237–252, 2003. doi: 10.1145/945445.945468. URL http://doi.acm.org/10.1145/945445.945468. [Pages 20 and 70]

[56] Facebook. Facebook LinkBench Benchmark. https://github.com/facebook/linkbench. Accessed: 2018-06-20. [Page 74]

[57] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. Unlocking energy. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, pages 393–406, 2016. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/falsafi. [Page 19]

[58] Bin Fan, David G. Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 371–384. USENIX Association, 2013. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan. [Page 25]

[59] Bin Fan, David G. Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 371–384, 2013. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan. [Pages 14, 20, and 56]

[60] Pascal Felber, Shady Issa, Alexander Matveev, and Paolo Romano. Hardware read-write lock elision. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 34:1–34:15, 2016. doi: 10.1145/2901318.2901346. URL http://doi.acm.org/10.1145/2901318.2901346. [Page 19]

[61] Michael Ferdman, Almutaz Adileh, Yusuf Onur Koçberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 37–48. ACM, 2012. doi: 10.1145/2150976.2150982. URL http://doi.acm.org/10.1145/2150976.2150982. [Page 35]

[62] Brad Fitzpatrick. Memcached. http://www.memcached.org, 2003. Accessed: 2018-06-20. [Page 74]

[63] Keir Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, UK, 2004. URL http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.599193. [Page 2]

[64] Vinitha Reddy Gankidi, Nikhil Teletia, Jignesh M. Patel, Alan Halverson, and David J. DeWitt. Indexing HDFS data in PDW: splitting the data from the index. *PVLDB*, 7(13): 1520–1528, 2014. URL http://www.vldb.org/pvldb/vol7/p1520-gankidi.pdf. [Page 21]

[65] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, pages 32:1–32:14, 2015. doi: 10.1145/2741948.2741973. URL http://doi.acm.org/10.1145/2741948.2741973. [Pages 20 and 56]

[66] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. Multicore locks: The case is not closed yet. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, pages 649–662, 2016. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/guiroux. [Pages 14, 19, 55, 56, and 100]

[67] Bart Haagdorens, Tim Vermeiren, and Marnix Goossens. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *Information Security Applications, 5th International Workshop, WISA 2004, Jeju Island, Korea, August 23-25, 2004, Revised Selected Papers*, pages 188–203. Springer, 2004. doi: 10.1007/978-3-540-31815-6_16. URL http://dx.doi.org/10.1007/978-3-540-31815-6_16. [Page 33]

[68] Bijun He, William N. Scherer III, and Michael L. Scott. Preemption adaptivity in time-published queue-based spin locks. In *High Performance Computing - HiPC 2005, 12th International Conference, Goa, India, December 18-21, 2005, Proceedings*, pages 7–18, 2005. doi: 10.1007/11602569_6. URL https://doi.org/10.1007/11602569_6. [Pages 19 and 64]

[69] Armin Heindl and Gilles Pokam. An analytic model for optimistic STM with lazy locking. In *Analytical and Stochastic Modeling Techniques and Applications, 16th International Conference, ASMTA 2009, Madrid, Spain, June 9-12, 2009. Proceedings*, pages 339–353. Springer, 2009. doi: 10.1007/978-3-642-02205-0_24. URL http://dx.doi.org/10.1007/978-3-642-02205-0_24. [Page 18]

[70] Armin Heindl and Gilles Pokam. An analytic framework for performance modeling of software transactional memory. *Computer Networks*, 53(8):1202–1214, 2009. doi: 10.1016/j.comnet.2009.02.006. URL http://dx.doi.org/10.1016/j.comnet.2009.02.006. [Pages 3, 18, and 25]

[71] Armin Heindl, Gilles Pokam, and Ali-Reza Adl-Tabatabai. An analytic model of optimistic software transactional memory. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA, Proceedings*, pages 153–162. IEEE Computer Society, 2009. doi: 10.1109/ISPASS.2009.4919647. URL http://dx.doi.org/10.1109/ISPASS.2009.4919647. [Page 18]

[72] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, pages 355–364, 2010. doi: 10.1145/1810479.1810540. URL http://doi.acm.org/10.1145/1810479.1810540. [Page 20]

[73] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012. ISBN 978-0-12-383872-8. [Page 9]

[74] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture. San Diego, CA, May 1993*, pages 289–300. ACM, 1993. doi: 10.1145/165123.165164. URL http://doi.acm.org/10.1145/165123.165164. [Page 11]

[75] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture. San Diego, CA, May 1993*, pages 289–300, 1993. doi: 10.1145/165123.165164. URL http://doi.acm.org/10.1145/165123.165164. [Page 20]

[76] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008. ISBN 978-0-12-370591-4. [Page 2]

[77] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA*, pages 522–529, 2003. doi: 10.1109/ICDCS.2003.1203503. URL https://doi.org/10.1109/ICDCS.2003.1203503. [Page 2]

[78] Adolfy Hoisie, Olaf M. Lubeck, and Harvey J. Wasserman. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *IJHPCA*, 14(4):330–346, 2000. doi: 10.1177/109434200001400405. URL http://dx.doi.org/10.1177/109434200001400405. [Page 18]

[79] Jian Huang, Xiangyong Ouyang, Jithin Jose, Md. Wasi-ur-Rahman, Hao Wang, Miao Luo, Hari Subramoni, Chet Murthy, and Dhabaleswar K. Panda. High-performance design of HBase with RDMA over infiniband. In *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*, pages 774–785, 2012. doi: 10.1109/IPDPS.2012.74. URL https://doi.org/10.1109/IPDPS.2012.74. [Page 15]

[80] Intel. An Introduction to the Intel QuickPath Interconnect. 2009. [Page 10]

[81] Intel. Transactional Synchronization Extensions Overview. https://software.intel.com/en-us/node/524022, 2013. Accessed: 2018-06-20. [Page 3]

[82] Intel. Intel 64 and ia-32 architectures software developer's manual, volume 3b: System programming guide. *Part*, 1:64, 2016. [Pages 27, 35, and 36]

[83] Engin Ipek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. An approach to performance prediction for parallel applications. In *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September 2, 2005, Proceedings*, pages 196–205. Springer, 2005. doi: 10.1007/11549468_24. URL http://dx.doi.org/10.1007/11549468_24. [Page 17]

[84] Raj Jain. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling*. Wiley professional computing. Wiley, 1991. ISBN 978-0-471-50336-1. [Pages 18 and 25]

[85] Víctor Jiménez, Francisco J. Cazorla, Roberto Gioiosa, Mateo Valero, Carlos Boneti, Eren Kursun, Chen-Yong Cher, Canturk Isci, Alper Buyuktosunoglu, and Pradip Bose. Power and thermal characterization of POWER6 system. In *19th International Conference on Parallel Architecture and Compilation Techniques (PACT 2010), Vienna, Austria, September 11-15, 2010*, pages 7–18. ACM, 2010. doi: 10.1145/1854273.1854281. URL http://doi.acm.org/10.1145/1854273.1854281. [Page 18]

[86] Lei Jin. Avoid Expensive Locks in Get(). https://rocksdb.org/blog/2014/06/27/avoid-expensive-locks-in-get.html, 2014. Accessed: 2018-06-20. [Page 20]

[87] Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mowry. Decoupling contention management from scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, pages 117–128, 2010. doi: 10.1145/1736020.1736035. URL http://doi.acm.org/10.1145/1736020.1736035. [Page 19]

[88] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 185–201, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia. [Pages 2, 15, 21, and 79]

[89] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 437–450, 2016. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia. [Page 15]

[90] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008. URL http://www.vldb.org/pvldb/1/1454211.pdf. [Page 15]

[91] Yagiz Kargin, Martin L. Kersten, Stefan Manegold, and Holger Pirk. The DBMS - your big data sommelier. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1119–1130, 2015. doi: 10.1109/ICDE.2015.7113361. URL https://doi.org/10.1109/ICDE.2015.7113361. [Page 21]

[92] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles, SOSP 1991, Asilomar Conference Center, Pacific Grove, California, USA, October 13-16, 1991*, pages 41–55, 1991. doi: 10.1145/121132.286599. URL http://doi.acm.org/10.1145/121132.286599. [Page 19]

[93] Darren J. Kerbyson, Henry J. Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J. Wasserman, and Michael L. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing, Denver, CO, USA, November 10-16, 2001, CD-ROM*, page 37. ACM, 2001. doi: 10.1145/582034.582071. URL http://doi.acm.org/10.1145/582034.582071. [Pages 18 and 25]

[94] Eric Koskinen and Maurice Herlihy. Dreadlocks: efficient deadlock detection. In *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008*, pages 297–303, 2008. doi: 10.1145/1378533.1378585. URL http://doi.acm.org/10.1145/1378533.1378585. [Page 20]

[95] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009. URL http://www.vldb.org/pvldb/2/vldb09-759.pdf. [Page 20]

[96] Eva Kwan, Sam Lightstone, Adam Storm, and Leanne Wu. Automatic configuration for ibm db2 universal database. In *Proc. of IBM Perf Technical Report*, 2002. [Page 21]

[97] FAL Labs. Kyoto Cabinet. http://fallabs.com/kyotocabinet. Accessed: 2018-06-20. [Pages 74 and 75]

[98] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007*, pages 249–258. ACM, 2007. doi: 10.1145/1229428.1229479. URL http://doi.acm.org/10.1145/1229428.1229479. [Pages 17 and 25]

[99] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 2–13, 2009. doi: 10.1145/1555754.1555758. URL http://doi.acm.org/10.1145/1555754.1555758. [Page 14]

[100] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. Accelerating relational databases by leveraging remote memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 355–370, 2016. doi: 10.1145/2882903.2882949. URL http://doi.acm.org/10.1145/2882903.2882949. [Page 15]

[101] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994.*, pages 25–35, 1994. doi: 10.1145/195473.195490. URL http://doi.acm.org/10.1145/195473.195490. [Page 19]

[102] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 429–444. USENIX Association, 2014. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim. [Page 25]

[103] Kevin T. Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin servers with smart pipes: designing soc accelerators for memcached. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pages 36–47, 2013. doi: 10.1145/2485922.2485926. URL http://doi.acm.org/10.1145/2485922.2485926. [Page 74]

[104] Rick Lindsley and Dave Hansen. Bkl: One Lock to Bind Them All. In *Ottawa Linux Symposium*, pages 301–309, 2002. [Pages 14 and 55]

[105] Xu Liu and Bo Wu. Scaanalyzer: a tool to identify memory scalability bottlenecks in parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 47:1–47:12. ACM, 2015. doi: 10.1145/2807591.2807648. URL http://doi.acm.org/10.1145/2807591.2807648. [Page 18]

[106] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. On the design and scalability of distributed shared-data databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 663–676, 2015. doi: 10.1145/2723372.2751519. URL http://doi.acm.org/10.1145/2723372.2751519. [Pages 2, 15, 21, and 79]

[107] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia L. Lawall, and Gilles Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 65–76, 2012. URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/lozi. [Page 20]

[108] Victor Luchangco, Daniel Nussbaum, and Nir Shavit. A hierarchical CLH queue lock. In *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference, Dresden, Germany, August 28 - September 1, 2006, Proceedings*, pages 801–810, 2006. doi: 10.1007/11823285_84. URL https://doi.org/10.1007/11823285_84. [Page 19]

[109] Gabriel Marin and John M. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004, June 10-14, 2004, New York, NY, USA*, pages 2–13. ACM, 2004. doi: 10.1145/1005686.1005691. URL http://doi.acm.org/10.1145/1005686.1005691. [Pages 3, 18, and 25]

[110] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: a lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 168–183, 2015. doi: 10.1145/2815400.2815406. URL http://doi.acm.org/10.1145/2815400.2815406. [Pages 20 and 56]

[111] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991. doi: 10.1145/103727.103729. URL http://doi.acm.org/10.1145/103727.103729. [Pages 12, 19, 57, and 59]

[112] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 267–275, 1996. doi: 10.1145/248052.248106. URL http://doi.acm.org/10.1145/248052.248106. [Page 63]

[113] Micron. 3D XPoint Technology, 2017. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology. [Page 14]

[114] Microsoft. OCS Open CloudServer Power Supply v2.0, 2015. http://www.opencompute.org/wiki/Server/SpecsAndDesigns. [Page 14]

[115] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: stanford transactional applications for multi-processing. In *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*, pages 35–46. IEEE Computer Society, 2008. doi: 10.1109/IISWC.2008.4636089. URL http://dx.doi.org/10.1109/IISWC.2008.4636089. [Pages 27 and 37]

[116] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 103–114, 2013. URL https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell. [Page 15]

[117] Mohamed Mohamedin, Roberto Palmieri, Sebastiano Peluso, and Binoy Ravindran. On designing numa-aware concurrency control for scalable transactional memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 45:1–45:2, 2016. doi: 10.1145/2851141.2851189. URL http://doi.acm.org/10.1145/2851141.2851189. [Page 20]

[118] Gordon E Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8):114, 1965. [Page 1]

[119] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 386–396, 2009. doi: 10.1109/ICSE.2009.5070538. URL https://doi.org/10.1109/ICSE.2009.5070538. [Page 20]

[120] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. Telecom Application Transaction Processing Benchmark. http://tatpbenchmark.sourceforge.net/. [Pages 15, 79, and 89]

[121] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 385–398. USENIX Association, 2013. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala. [Page 25]

[122] Graham R. Nudd, Darren J. Kerbyson, Efstathios Papaefstathiou, S. C. Perry, John S. Harper, and Daniel V. Wilcox. Pace - A toolset for the performance prediction of parallel and distributed systems. *IJHPCA*, 14(3):228–251, 2000. doi: 10.1177/109434200001400306. URL http://dx.doi.org/10.1177/109434200001400306. [Page 25]

[123] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. Slalom: Coasting through raw data via adaptive partitioning and indexing. *PVLDB*, 10(10):1106–1117, 2017. doi: 10.14778/3115404.3115415. URL http://www.vldb.org/pvldb/vol10/p1106-olma.pdf. [Page 86]

[124] Catherine Rose Mills Olschanowsky. *Hpc Application Address Stream Compression, Replay and Scaling*. PhD thesis, University of California at San Diego, La Jolla, CA, USA, 2011. [Page 17]

[125] Oracle. MySQL. http://www.mysql.com. Accessed: 2018-06-20. [Page 74]

[126] João Paiva, Pedro Ruivo, Paolo Romano, and Luís E. T. Rodrigues. AUTOPLACER: scalable self-tuning data placement in distributed key-value stores. In *10th International Conference on Autonomic Computing, ICAC'13, San Jose, CA, USA, June 26-28, 2013*, pages 119–131, 2013. URL https://www.usenix.org/conference/icac13/technical-sessions/presentation/paiva. [Page 20]

[127] Stratos Papadomanolakis and Anastassia Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM 2004), 21-23 June 2004, Santorini Island, Greece*, pages 383–392, 2004. doi: 10.1109/SSDBM.2004.19. URL http://doi.ieeecomputersociety.org/10.1109/SSDBM.2004.19. [Page 86]

[128] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual Symposium on Computer Architecture, Ann Arbor, USA, June 1984*, pages 348–354, 1984. doi: 10.1145/800015.808204. URL http://doi.acm.org/10.1145/800015.808204. [Page 9]

[129] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 61–72, 2012. doi: 10.1145/2213836.2213844. URL http://doi.acm.org/10.1145/2213836.2213844. [Pages 20 and 22]

[130] Eleni Petraki, Stratos Idreos, and Stefan Manegold. Holistic indexing in main-memory column-stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1153–1166, 2015. doi: 10.1145/2723372.2723719. URL http://doi.acm.org/10.1145/2723372.2723719. [Page 21]

[131] Carl Adam Petri. Communication with automata, new york: Griffiss air force base. Technical report, Tech. Rep. RADC-TR-65-377, 1966. [Page 18]

[132] James R. Phillips. Zunzun.com, 2013. URL http://www.zunzun.com. [Page 34]

[133] Danica Porobic, Erietta Liarou, Pinar Tözün, and Anastasia Ailamaki. Atrapos: Adaptive transaction processing on hardware islands. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 688–699, 2014. doi: 10.1109/ICDE.2014.6816692. URL https://doi.org/10.1109/ICDE.2014.6816692. [Page 86]

[134] Donald E. Porter and Emmett Witchel. Understanding transactional memory performance. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, 28-30 March 2010, White Plains, NY, USA*, pages 97–108. IEEE Computer Society, 2010. doi: 10.1109/ISPASS.2010.5452061. URL http://dx.doi.org/10.1109/ISPASS.2010.5452061. [Page 18]

[135] Hari K. Pyla and Srinidhi Varadarajan. Avoiding deadlock avoidance. In *19th International Conference on Parallel Architecture and Compilation Techniques, PACT 2010, Vienna, Austria, September 11-15, 2010*, pages 75–86, 2010. doi: 10.1145/1854273.1854288. URL http://doi.acm.org/10.1145/1854273.1854288. [Page 20]

[136] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. SWORD: scalable workload-aware data placement for transactional workloads. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 430–441, 2013. doi: 10.1145/2452376.2452427. URL http://doi.acm.org/10.1145/2452376.2452427. [Pages 20 and 22]

[137] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 24–33, 2009. doi: 10.1145/1555754.1555760. URL http://doi.acm.org/10.1145/1555754.1555760. [Page 14]

[138] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture, Austin, Texas, USA, December 1-5, 2001*, pages 294–305, 2001. doi: 10.1109/MICRO.2001.991127. URL https://doi.org/10.1109/MICRO.2001.991127. [Page 19]

[139] Amitabha Roy, Steven Hand, and Timothy L. Harris. A runtime system for software lock elision. In *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, pages 261–274, 2009. doi: 10.1145/1519065.1519094. URL http://doi.acm.org/10.1145/1519065.1519094. [Page 19]

[140] Christoph Rupp. HamsterDB. http://hamsterdb.com. Accessed: 2018-06-20. [Pages 74 and 75]

[141] Jim Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995. doi: 10.1006/jagm.1995.1021. URL http://dx.doi.org/10.1006/jagm.1995.1021. [Page 39]

[142] Malavika Samak and Murali Krishna Ramanathan. Trace driven dynamic deadlock detection and reproduction. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 29–42, 2014. doi: 10.1145/2555243.2555262. URL http://doi.acm.org/10.1145/2555243.2555262. [Page 20]

[143] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *PVLDB*, 10(4):445–456, 2016. URL http://www.vldb.org/pvldb/vol10/p445-serafini.pdf. [Pages 20 and 22]

[144] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, pages 204–213. ACM, 1995. doi: 10.1145/224964.224987. URL http://doi.acm.org/10.1145/224964.224987. [Page 11]

[145] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, pages 204–213, 1995. doi: 10.1145/224964.224987. URL http://doi.acm.org/10.1145/224964.224987. [Page 20]

[146] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 317–332, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/shi. [Page 15]

[147] Karan Singh, Major Bhadauria, and Sally A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Computer Architecture News*, 37 (2):46–55, 2009. doi: 10.1145/1577129.1577137. URL http://doi.acm.org/10.1145/1577129.1577137. [Page 18]

[148] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. Coherence stalls or latency tolerance: Informed CPU scheduling for socket and core sharing. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 323–336. USENIX Association, 2016. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/srikanthan. [Page 18]

[149] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1150–1160, 2007. URL http://www.vldb.org/conf/2007/papers/industrial/p1150-stonebraker.pdf. [Page 14]

[150] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008. [Page 14]

[151] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. Wimpy nodes with 10GbE: Leveraging one-sided operations in soft-RDMA to boost memcached. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 347–353, 2012. URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/stuedi. [Page 15]

[152] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. Fine-grained partitioning for aggressive data skipping. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1115–1126, 2014. doi: 10.1145/2588555.2610515. URL http://doi.acm.org/10.1145/2588555.2610515. [Page 22]

[153] Sun Microsystems. Multithreading in the Solaris Operating Environment. Technical report, 2002. [Page 19]

[154] SQLite Development Team. SQLite. http://sqlite.org. Accessed: 2018-06-20. [Page 74]

[155] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 1–12, 2012. doi: 10.1145/2213836.2213838. URL http://doi.acm.org/10.1145/2213836.2213838. [Page 15]

[156] Josep Torrellas, Yan Solihin, and Vinh Vi Lam. Scal-tool: Pinpointing and quantifying scalability bottlenecks in DSM multiprocessors. In *Proceedings of the ACM/IEEE Conference on Supercomputing, SC 1999, November 13-19, 1999, Portland, Oregon, USA*, page 17. ACM, 1999. doi: 10.1145/331532.331549. URL http://doi.acm.org/10.1145/331532.331549. [Page 18]

[157] Dinh Nguyen Tran, Phung Chinh Huynh, Y. C. Tay, and Anthony K. H. Tung. A new approach to dynamic self-tuning of database buffers. *TOS*, 4(1):3:1–3:25, 2008. doi: 10.1145/1353452.1353455. URL http://doi.acm.org/10.1145/1353452.1353455. [Page 21]

[158] Khai Q. Tran, Jeffrey F. Naughton, Bruhathi Sundarmurthy, and Dimitris Tsirogiannis. JECB: a join-extension, code-based approach to OLTP data partitioning. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 39–50, 2014. doi: 10.1145/2588555.2610532. URL http://doi.acm.org/10.1145/2588555.2610532. [Page 22]

[159] Transaction Processing Performance Council (TPC). TPC Benchmark C: Standard specification. http://www.tpc.org/tpcc/. [Page 94]

[160] Peter Triantafillou. An approach to deadlock detection in multidatabases. *Inf. Syst.*, 22(1):39–55, 1997. doi: 10.1016/S0306-4379(97)00003-3. URL https://doi.org/10.1016/S0306-4379(97)00003-3. [Page 20]

[161] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 306–324, 2017. doi: 10.1145/3132747.3132762. URL http://doi.acm.org/10.1145/3132747.3132762. [Page 15]

[162] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95, Santa Margherita Ligure, Italy, June 22-24, 1995*, pages 392–403, 1995. doi: 10.1145/223982.224449. URL http://doi.acm.org/10.1145/223982.224449. [Page 10]

[163] Takayuki Usui, Reimer Behrends, Jacob Evans, and Yannis Smaragdakis. Adaptive locks: Combining transactions and locks for efficient concurrency. In *PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 12-16 September 2009, Raleigh, North Carolina, USA*, pages 3–14. IEEE Computer Society,

2009. doi: 10.1109/PACT.2009.20. URL http://dx.doi.org/10.1109/PACT.2009.20. [Page 18]

[164] Augusto Vega, Alper Buyuktosunoglu, and Pradip Bose. Smt-centric power-aware thread placement in chip multiprocessors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*, pages 167–176. IEEE Computer Society, 2013. doi: 10.1109/PACT.2013.6618814. URL http://dx.doi.org/10.1109/PACT.2013.6618814. [Page 18]

[165] Hannes Voigt, Thomas Kissinger, and Wolfgang Lehner. SMIX: self-managing indexes for dynamic workloads. In *Conference on Scientific and Statistical Database Management, SSDBM '13, Baltimore, MD, USA, July 29 - 31, 2013*, pages 24:1–24:12, 2013. doi: 10.1145/2484838.2484862. URL http://doi.acm.org/10.1145/2484838.2484862. [Page 21]

[166] Richard West, Puneet Zaroo, Carl A. Waldspurger, and Xiao Zhang. Online cache modeling for commodity multicore processors. In *19th International Conference on Parallel Architecture and Compilation Techniques (PACT 2010), Vienna, Austria, September 11-15, 2010*, pages 563–564. ACM, 2010. doi: 10.1145/1854273.1854353. URL http://doi.acm.org/10.1145/1854273.1854353. [Pages 18 and 29]

[167] Amy L. Williams, William Thies, and Michael D. Ernst. Static deadlock detection for java libraries. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, pages 602–629, 2005. doi: 10.1007/11531142_26. URL https://doi.org/10.1007/11531142_26. [Page 20]

[168] Eugene Wu and Samuel Madden. Partitioning techniques for fine-grained indexing. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 1127–1138, 2011. doi: 10.1109/ICDE.2011.5767830. URL https://doi.org/10.1109/ICDE.2011.5767830. [Page 86]

[169] Lingxiang Xiang and Michael L. Scott. Software partitioning of hardware transactions. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 76–86, 2015. doi: 10.1145/2688500.2688506. URL http://doi.acm.org/10.1145/2688500.2688506. [Page 20]

[170] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining ACID and BASE in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 495–509, 2014. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/xie. [Page 15]

[171] Khaled Yagoub, Peter Belknap, Benoît Dageville, Karl Dias, Shantanu Joshi, and Hailing Yu. Oracle's SQL performance analyzer. *IEEE Data Eng. Bull.*, 31(1):51–58, 2008. URL http://sites.computer.org/debull/A08mar/yagoub.pdf. [Page 21]

[172] Leo T. Yang, Xiaosong Ma, and Frank Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA, CD-Rom*, page 40. IEEE Computer Society, 2005. doi: 10.1109/SC.2005.20. URL http://dx.doi.org/10.1109/SC.2005.20. [Page 18]

[173] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pages 35–44. IEEE Computer Society, 2014. doi: 10.1109/ISPASS.2014.6844459. URL http://dx.doi.org/10.1109/ISPASS.2014.6844459. [Page 18]

[174] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. The end of a myth: Distributed transaction can scale. *PVLDB*, 10(6):685–696, 2017. URL http://www.vldb.org/pvldb/vol10/p685-zamanian.pdf. [Pages 2, 15, 21, 79, and 86]

[175] Jidong Zhai, Wenguang Chen, and Weimin Zheng. PHANTOM: predicting performance of parallel applications on large-scale parallel machines using a single node. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pages 305–314. ACM, 2010. doi: 10.1145/1693453.1693493. URL http://doi.acm.org/10.1145/1693453.1693493. [Pages 18 and 25]

[176] Mingzhe Zhang, Francis C. M. Lau, Cho-Li Wang, Luwei Cheng, and Haibo Chen. Scalable adaptive numa-aware lock: combining local locking and remote locking for efficient concurrency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 50:1–50:2, 2016. doi: 10.1145/2851141.2851176. URL http://doi.acm.org/10.1145/2851141.2851176. [Page 19]

[177] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 design advisor: Integrated automatic physical database design. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 1087–1097, 2004. URL http://www.vldb.org/conf/2004/IND4P1.PDF. [Page 21]

# Georgios Chatzopoulos

*Curriculum Vitae*

*EPFL IC IINFCOM DCL INR 312*
*Station 14*
*1015 Lausanne*
☎ *+41 21 693 8121*
✉ *georgios.chatzopoulos@epfl.ch*
*people.epfl.ch/georgios.chatzopoulos*

*"Citius, Altius, Fortius"*

## Research Interests

My research interest lies in the area of synchronization in parallel and distributed systems. I am currently looking into how to improve widely used techniques of synchronization, such as locking and distributed transactions, on modern hardware. In the past, I have also worked on performance and scalability analysis for in-memory applications.

## Education

2013 – **PhD in Computer and Communication Sciences**, EPFL, Switzerland.
- Dissertation: *"Scalable Synchronization in Shared-Memory Systems: Extrapolating, Adapting, Tuning"*.
- Advisor: Rachid Guerraoui.

2008 - 2012 **Diploma in Electrical and Computer Engineering (M.Eng. Equivalent)**, NTUA, Greece.
- GPA: 9.20/10.00 – ranked 11$^{th}$ in graduation year (out of 330 students).
- Major/Minor: Computer Science / Networks – GPA in major: 9.79/10.00.
- Thesis: "Scalability Prediction for Parallel Regions in Multicore Architectures".
- Advisor: Nectarios Koziris.

## Publications

2018 G. Chatzopoulos, A. Dragojević and R. Guerraoui. "*SPADE: Tuning scale-out OLTP on modern RDMA clusters*". In the Proceedings of the 19th International Middleware Conference (Middleware '18), 2018 (to appear).

2017 G. Chatzopoulos, A. Dragojević and R. Guerraoui. *"ESTIMA: Extrapolating ScalabiliTy of In-Memory Applications"*. In ACM Transactions on Parallel Computing (TOPC) - Special Issue: Invited papers from PPoPP 2016, Part 2, 2017.

2017 G. Chatzopoulos, R. Guerraoui, T. Harris, and V. Trigonakis. "*Abstracting Multi-Core Topologies with MCTOP*". In the Proceedings of the 12th European Conference on Computer Systems (EuroSys '17), 2017.

2016 J. Antić, G. Chatzopoulos, R. Guerraoui, and V. Trigonakis. *"Locking Made Easy"*. In the Proceedings of the 17th International Middleware Conference (Middleware '16), 2016.

2016 G. Chatzopoulos, A. Dragojević and R. Guerraoui. *"ESTIMA: Extrapolating ScalabiliTy of In-Memory Applications"*. In the Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16), 2016.

2013 G. Chatzopoulos. K. Kourtis, N. Koziris, G. Goumas. *"Towards a compiler/runtime synergy to predict the scalability of parallel loops"*. In the Proceedings of the 6th IEEE International Workshop on Multi-/Many-core Computing Systems (MuCoCoS 2013).

## Professional Experience

2015 **Research intern**, Microsoft Research, Cambridge, UK.
- Part of the FaRM project team.
- Implemented new protocols for distributed transactions in the datacenter.

2012–2013 **Software Engineer**, Kalahari Ltd., Woking, UK.
- Gathered and analyzed user requirements.
- Implemented new features for an FX pricing and publishing platform.
- Developed testing infrastructure for new and existing functionality.

## Extracurricular Experience

2015– **Global Director**, ShARE Student Strategy Consulting.
- Responsible for output quality and knowledge transfer across 31 universities and 500+ students globally.
- Led a team of 6 people, assessing more than 100 junior consulting presentations.
- Implemented efficient progress tracking processes for teams.

2013– **Senior Member, Manager, Vice President (2015)**, ShARE EPFL, Switzerland.
- Participated and managed consulting projects with start-up clients.
- Negotiated and secured collaborations with corporations and start-ups.
- Organized consulting workshops for members with consulting partners.

## Talks and Presentations

2018 *"Tuning scale-out OLTP on modern RDMA clusters"*, EPFL.

2017 *"Revisiting Transactional Computing on Modern Hardware"*, Microsoft Research Cambridge.

2016 *"Locking Made Easy"*, Middleware Conference.

2016 *"Extrapolating the scalability of in-memory applications"*, PPoPP Conference.

## Service

2017 Shadow Program Committee member for EuroSys '17.

## Grants and Awards

2016 Microsoft Research Swiss Joint Research Centre Project: "*Revisiting Transactional Computing on Modern Hardware*".

2016 Best Paper Award for the paper "*Locking Made Easy*", Middleware '16.

2013 École Polytechnique Fédérale de Lausanne (EPFL) EDIC Fellowship.

## In The News

2016 **IC researchers gain recognition**, EPFL IC News.

2016 **Six IC projects gain funding from Microsoft**, EPFL IC News.

## Teaching Experience

- **Concurrent algorithms:** Master Course, 2014-2016.
- **Programming I:** Bachelor Course, 2016-2017.
- **Programming II:** Bachelor Course, 2014-2017.
- **Information, Computation and Communication:** Bachelor Course, 2017-2018.

## Languages

Greek **Native**

English **Full professional proficiency** *C2 Level*

French **Intermediate knowledge** *B2 Level*

## Interests

- Ice skating - Skiing - Electric guitar